Python Tutorial

Release 2.7

March Liu<march.liu@gmail.com>

October 26, 2010

CONTENTS

Release 2.7

Date October 09, 2010

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Python 是一门简单易学的强大编程语言。它有高效的高级数据结构和简单有效的 面向对象编程方式。Python 的优雅语法和动态类型，以及它解释语言的特质， 使其在很多平台上都成为理想的脚本和快速应用开发语言。

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, http://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

Python 解释器及其扩展标准库的源码和编译版本可以从 Python 的 Web 站点 http://www.python.org/ 免费获得，并且可以自由分发。该站点还提供了很多 免费的第三方 Python 模块、程序和工具，及其附加文档。

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

Python 解释器可以很容易的通过 C 或者 C++ （或其它可以通过 C 调用的语 言）扩展新的函数和数据类型。Python 也可以作为定制应用的扩展语言。

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

本手册向读者介绍 Python 语言及其系统的基本知识与概念。配合 Python 解释 器学习会很有帮助，不过所有的例子都已经包括在文中，所以这本手册也可以离 线阅读。

For a description of standard objects and modules, see library-index. reference-index gives a more formal definition of the language. To write extensions in C or C++, read extending-index and c-api-index. There are also several books covering Python in depth.

标准对象和模块的詳細介绍请参见 library-index 。 reference-index 提供了更多语言方面的说明。如编写 C 或 C++ 的 扩展，请阅读 extending-index 和 c-api-index 。这些文档深 入介绍了相关的 Python 知识。

This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python's most noteworthy features, and will give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in library-index.

本手册不会涵盖 Python 的所有功能，也不会解释所用到的所有相关知识。相 反，它介绍了许多 Python 中引人注目的功能，这会对读者掌握这门语言的风 格大有帮助。读过它后，你应该可以阅读和编写 Python 模块和程序，接下来可 以从 library-index 中进一步学习Python复杂多变的库和模块。

The glossary is also worth going through.

glossary 也值得一读。

CHAPTER

ONE

WHETTING YOUR APPETITE 开胃菜

If you do much work on computers, eventually you find that there's some task you'd like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you'd like to write a small custom database, or a specialized GUI application, or a simple game.

如果你要在计算机上做大量的工作，希望它们能够自动化一些。例如，你想在大 量的文本文件中进行查找替换，或者通过复杂的方式将一批图像文件重命名，修 改尺寸。可能你想写一个小型的定制数据库，或者特定的 GUI 应用程序，或者 一个简单的游戏。

If you're a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you're writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you've written a program that could use an extension language, and you don't want to design and implement a whole new language for your application.

如果你是专业的软件开发者，可能会想使用一些 C/C++/Java 库，但是发现通常 的编写/编译/测试/重编译周期太慢了。也许你想为每个写一个测试，可这样做 太麻烦。或者你需要写一个带有扩展语言的程序，可不想为你的应用程序设计和 实现一个新的语言。

Python is just the language for you.

Python 就是你需要的语言。

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, Mac OS X, and Unix operating systems, and will help you get the job done more quickly.

你也可以写一个 Unix shell 脚本或者 Windows 批处理文件完成任务，不过 shell 脚本最擅长移动文件和修改文本数据，不擅长编写图形界面的应用程序或 游戏。你可以写个 C/C++/Java 程序，但是就算是一个最简单的草案，也要花 费太多的开发时间。 Python 更易用，并且在 Windows， Mac OS X 和 Unix 操 作系统上都可用，能够帮你更快的完成任务。

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python 简单易用，但它是真正的编程语言，比起 shell 脚本或批处理文件，提 供了更多的结构和对大型程序的支持。另一方面，Python 比起C，提供更多的错 误检查。而且作为 非常高级的编程语言 ，它内置

了类似变长数组和字典这样 的高级数据结构。Python 提供了更为通用的数据类型，所以它比 Awk 甚至 Perl 适合更广大的问题领域，做其它的很多事，Python 至少也不会比别的编程语言更复杂。

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs --- or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python 可以让你把自己的程序分隔成不同的模块，以便在其它的 Python 程序 中重用。这样你就可以让自己的程序基于一个很大的标准模块集，也可以把它们作为 学习 Python 编程的示例。这些模块包括文件 I/O，系统调 用，sockets，甚至像 Tk 这样的图形工具接口。

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python是一门解释型语言，因为不需要编译和链接的时间，它可以帮你省下一些 开发时间。解释器可以交互式使用，这样就可以很方便的测试语言中的各种功 能，以便于编写一次性的临时程序，或者进行自下而上的开发。还可以当它是一个随 手可用的计算器。

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

Python 可以写出很紧凑，可读性很强的程序。用 Python 写的程序通常比同样 的 C、C++ 或Java程序要短得多，这是因为以下几个原因：

- the high-level data types allow you to express complex operations in a single statement;

  高级数据结构使你可以在一个单独的语句中表达出很复杂的操作；

- statement grouping is done by indentation instead of beginning and ending brackets;

  语句的组织依赖于缩进而不是 begin/end 标识；

- no variable or argument declarations are necessary.

  不需要变量或参数声明。

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

Python 是 可扩展的：如果你会用 C 语言写程序，就可以很容易的为解释器 添加新的内置模块和功能，或者优化瓶颈，使其达到最大速度，或者使 Python 能够链接到某些只以二进制形式提供的库（比如某个专用的商业图形库）。等你眞正熟悉 这一切了，你就可以将 Python 集成进由 C 写成的程序，把 Python 当做这个 程序的扩展或命令行语言。

By the way, the language is named after the BBC show ``Monty Python's Flying Circus'' and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

顺便说一下，这个语言的名字来源于 BBC 的“Monty Python's Flying Circus”节目，和凶猛的爬虫没有任何关系。在文档中引用 Monty Python 典故 不仅可以，而且还很恰当！

Now that you are all excited about Python, you'll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

现在你可能会觉得 Python 如此激动人心，想仔细的试试它了。 学习一门语言最好的办法就是使用它，如你所读到的，本文会引领你运用 Python 解释器。

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

下一节中，我们将会说明解释器的用法。这没有什么神秘的内容，不过有助于我 们练习后面展示的例子。

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.

本指南其它部分通过例子介绍了 Python 语言和系统的各种功能，开始是简单表 达式、语法和数据类型，接下来是函数和模块，最后是诸如异常和自定义类这样 的高级内容。

CHAPTER

TWO

USING THE PYTHON INTERPRETER 使用 PYTHON 解释器

2.1 Invoking the Interpreter 调用解释器

The Python interpreter is usually installed as /usr/local/bin/python on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command

通常 Python 的解释器被安装在目标机器的 /usr/local/bin/python 目录下； 把 /usr/local/bin 目录放进你的 Unix Shell 的搜索路径里，确保它可以通过 输入

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)

来启动。因为安装路径是可选的，所以也有可能安装在其它位置，你可以与安装 Python 的用户或系统管理员联系。（例如， /usr/local/python 就是一个很常见 的选择）

On Windows machines, the Python installation is usually placed in C:\Python27, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box

在 Windows机器上，Python 通常安装在 C:\Python27 当然，我们在运行安装程序的 时候可以改变它。需要把这个目录加入到我们的 Path 中的话，可以像下面这样 在 DOS 窗中输入命令行

```
set path=%path%;C:\\python27
```

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: quit().

输入一个文件结束符（Unix上是 Ctrl+D ，Windows上是 Ctrl+Z ）解释器会以 0 值退出。如果这没有起作用，你可以输入以下命令退出： quit() 。

The interpreter's line-editing features usually aren't very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input Editing and History Substitution for an introduction to the keys. If nothing appears to happen, or if ^P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

解释器的行编辑功能并不很复杂。装在 Unix 上的解释器可能会有 GNU readline 库支持，这样就可以额外得到精巧的交互编辑和历史记录功能。可能检查命令行 编辑器支持能力。最方便的方式是在主提示符下输入 Control-P。如果有嘟嘟声（计算 机扬声器），说明你可以使用命令行编辑功能，从附录 Interactive Input Editing and History Substitution 可以查到快捷键的 介绍。如果什么声音也没有，或者显示 ^p ，说明命令行编辑功能不可用， 你只有用退格键删掉输入的命令了。

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.

解释器的操作有些像 Unix Shell：使用终端设备做为标准输入来调用它时，解 释器交互的解读和执行命令，通过文件名参数或以文件做为标准输入设备时，它 从文件中解读并执行 脚本 。

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in command, analogous to the shell's *-c* option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote command in its entirety with single quotes.

启动解释器的第二个方法是 `python -c command [arg] ...` ，这种方法可以在 命令行 中直接执行语句，等同于Shell的 -c 选项。因为Python语句通常会包括 空格之类的特殊字符，所以最好把整个 命令 用单引号包起来。

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for module as if you had spelled out its full name on the command line.

有些 Python 模块也可以当作脚本使用。它们可以用 `python -m module [arg]...` 调用，这样就会像你在命令行中给出其完整名字一样运行 模块 源文件。

Note that there is a difference between `python file` and `python <file`. In the latter case, input requests from the program, such as calls to `input()` and `raw_input()`, are satisfied from file. Since this file has already been read until the end by the parser before the program starts executing, the program will encounter end-of-file immediately. In the former case (which is usually what you want) they are satisfied from whatever file or device is connected to standard input of the Python interpreter.

注意 `python file` 和 `python <file` 是有区别的。对于后一种情况，程序中类 似于调用 `input()` 和 `raw_input()` 这样的输入请求，来自于确定 的 文件 。因为在解析器开始执行之前，文件已经完全读入，所以程序指向文件尾。在前一种情况 （这通常是你需要的） 它们读取联接到 Python 解释器的标准输入，无关它们是文件还是其它设备。

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing *-i* before the script. (This does not work if the script is read from standard input, for the same reason as explained in the previous paragraph.)

使用脚本文件时，经常会运行脚本然后进入交互模式。这也可以通过在脚本之前 加上 *-i* 参数来实现。（如果脚本来自标准输入，就不能这样运行，与前一段提到的原因一样。）

2.1.1 Argument Passing 参数传递

When known to the interpreter, the script name and additional arguments thereafter are passed to the script in the variable `sys.argv`, which is a list of strings. Its length is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When *-c* command is used, `sys.argv[0]` is set to `'-c'`. When *-m* module is used, `sys.argv[0]` is set to the full name of the located module. Options found after *-c* command or *-m* module are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

调用解释器时，脚本名和附加参数传入一个名为 `sys.argv` 的字符串列表。没有 给定脚本和参数时，它至少也有一个元素： `sys.argv[0]` 此时为空字符串。脚本 名指定为 `'-'` （表示标准输入）时，`sys.argv[0]` 被设定为 `'-'` ，使用 *-c* 指令时， `sys.argv[0]` 被设定为 `'-c'` 。 使用 *-m* module 参数时， `sys.agv[0]` 被设定 为指定模块的全名。:option:-c command 或者 *-m* module 之 后的参数不会被 Python 解释器的选项处理机制所截获，而是留在 `sys.argv` 中，供脚本命令操作。

2.1.2 Interactive Mode 交互模式

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (`>>>`); for continuation lines it prompts with the secondary prompt, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

从 tty 读取命令时，我们称解释器工作于 交互模式 。这种模式下它根据 主提示符 来执行，主提示符通常标识为三个大于号（ `>>>` ）；继续的部分被称为 从属提示符 ，由三个点标识（ `...` ）。在第一行之前，解释器打印欢迎信息、 版本号和授权提示

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

输入多行结构时需要从属提示符了，例如，下面这个 if 语句

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2.2 The Interpreter and Its Environment 解释器及其环境

2.2.1 Error Handling 错误处理

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

有错误发生时，解释器打印一个错误信息和栈跟踪器。交互模式下，它返回主提 示符，如果从文件输入执行，它在打印栈跟踪器后以非零状态退出。（异常可以 由 `try` 语句中的 `except` 子句来控制，这样就不会出现 上文中的错误信息）有一些非常致命的错误会导致非零状态下退出，这由通常由 内部矛盾和内存溢出造成。所有的错误信息都写入标准错误流；命令中执行的普 通输出写入标准输出。

Typing the interrupt character (usually Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt. [1] Typing an interrupt while a command is executing raises the KeyboardInterrupt exception, which may be handled by a try statement.

在主提示符或附属提示符输入中断符（通常是Control-C 或者 DEL）就会取消当 前输入，回到主命令行。[2] 2.2.执行命令时输入一个中断符会抛出一个 KeyboardInterrupt 异常，它可以被 try 句截获。

2.2.2 Executable Python Scripts 执行 Python 脚本

On BSD'ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line:

BSD类的 Unix系统中，Python 脚本可以像 Shell 脚本那样直接执行。只要在脚 本文件开头写一行命令，指定文件和模式

```
#! /usr/bin/env python
```

(assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode. The #! must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending ('\n'), not a Windows ('\r\n') line ending. Note that the hash, or pound, character, '#', is used to start a comment in Python.

(要确认 Python 解释器在用户的 PATH 中) #! 必须是文件的前 两个字符，在某些平台上，第一行必须以 Unix 风格的行结束符（ '\n' ）结束， 不能用 Windows （ '\r\n' ） 的结束符。注意， '#' 是Python 中是行 注释的起始符。

The script can be given an executable mode, or permission, using the chmod command:

脚本可以通过 chmod 命令指定执行模式和权限

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an ``executable mode''. The Python installer automatically associates .py files with python.exe so that a double-click on a Python file will run it as a script. The extension can also be .pyw, in that case, the console window that normally appears is suppressed.

Windows 系统上没有“执行模式”。 Python 安装程序自动将 .py 文件关 联到 python.exe ，所以在 Python 文件图标上双击，它就会作为脚本执行。 同样 .pyw 也作了这样的关联，通常它执行时不会显示控制台窗口。

2.2.3 Source Code Encoding 源程序编码

It is possible to use encodings different than ASCII in Python source files. The best way to do it is to put one more special comment line right after the #! line to define the source file encoding:

Python 的源文件可以通过编码使用 ASCII 以外的字符集。最好的做法是在 #! 行后面用一个特殊的注释行来定义字符集

```
# -*- coding: encoding -*-
```

With that declaration, all characters in the source file will be treated as having the encoding encoding, and it will be possible to directly write Unicode string literals in the selected

[1] A problem with the GNU Readline package may prevent

[2] 包的一个问题可能会造成它无法正常工作。

encoding. The list of possible encodings can be found in the Python Library Reference, in the section on codecs.

根据这个声明，Python 会尝试将文件中的字符编码转为 encoding 编码。并且，它 尽可能的将指定的编码直接写成 Unicode 文本。在 Python 库参考手册 中 codecs 部份可以找到可用的编码列表（推荐使用 utf-8 处理中文——译者注）。

For example, to write Unicode literals including the Euro currency symbol, the ISO-8859-15 encoding can be used, with the Euro symbol having the ordinal value 164. This script will print the value 8364 (the Unicode codepoint corresponding to the Euro symbol) and then exit:

例如，可以用 ISO-8859-15 编码可以用来编写包含欧元符号的 Unicode 文本， 其编码值为 164。这个脚本会输出 8364 （欧元符号的 Unicode 对应编码） 然后退出

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

If your editor supports saving files as UTF-8 with a UTF-8 byte order mark (aka BOM), you can use that instead of an encoding declaration. IDLE supports this capability if Options/General/Default Source Encoding/UTF-8 is set. Notice that this signature is not understood in older Python releases (2.2 and earlier), and also not understood by the operating system for script files with #! lines (only used on Unix systems).

如果你的文件编辑器可以将文件保存为 UTF-8 格式，并且可以保存 UTF-8 字节序标记 （即 BOM - Byte Order Mark），你可以用这个来代替编码声明。IDLE可以通过设定 Options/General/Default Source Encoding/UTF-8 来支持它。需要注意的是旧 版 Python 不支持这个标记（Python 2.2或更早的版本），也同样不去理解由操 作系统调用脚本使用的 #! 行（仅限于 Unix系统）。

By using UTF-8 (either through the signature or an encoding declaration), characters of most languages in the world can be used simultaneously in string literals and comments. Using non-ASCII characters in identifiers is not supported. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

使用 UTF-8 内码（无论是用标记还是编码声明），我们可以在字符串和注释中 使用世界上的大部分语言。标识符中不能使用非 ASCII 字符集。为了正确显示 所有的字符，你一定要在编辑器中将文件保存为 UTF-8 格式，而且要使用支持 文件中所有字符的字体。

2.2.4 The Interactive Startup File 交互式环境的启动文件

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named PYTHONSTARTUP to the name of a file containing your start-up commands. This is similar to the .profile feature of the Unix shells.

使用 Python 解释器的时候，我们可能需要在每次解释器启动时执行一些命令。 你可以在一个文件中包含你想要执行的命令，设定一个名为 PYTHONSTARTUP 的 环境变量来指定这个文件。这类似于 Unix shell的 .profile 文件。

This file is only read in interactive sessions, not when Python reads commands from a script, and not when /dev/tty is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts sys.ps1 and sys.ps2 in this file.

这个文件在交互会话期是只读的，当 Python 从脚本中解读文件或以终端 /dev/tty 做为外部命令源时则不会如此（尽管它们的行为很像是处在交互会话 期。）它与解释器执行的命令处在同一个命名空间，所以由它定义或引用的一切 可以在解释器中不受限制的使用。你也可以在这个文件中改变 sys.ps1 和 sys.ps2 指令。

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py'). If you want to use the startup file in a script, you must do this explicitly in the script

如果你想要在当前目录中执行附加的启动文件，可以在全局启动文件中加入类似 以下的代码： if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py') 。如果你想要在某个脚本中使用启动文件，必须要在脚本中写入这样的语句

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

CHAPTER

THREE

AN INFORMAL INTRODUCTION TO PYTHON PYTHON 概要介绍

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

下面的例子中，输入和输出分别由大于号和句号提示符（ >>> 和 `...` ）标注。如果想重现这些例子，就要在解释器的提示符后，输入（提示 符后面的）那些不包含提示符的代码行。需要注意的是在练习中遇到的从属提示符表示 你需要在最后多输入一个空行，解释器才能知道这是一个多行命令的结束。

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

本手册中的很多示例——包括那些带有交互提示符的——都含有注释。Python 中的 注释以 # 字符起始，直至实际的行尾（译注——这里原作者用了 physical line 以表示实际的换行而非编辑器的自动换行）。注释可以从行 首开始，也可以在空白或代码之后，但是不出现在字符串中。文本字符串中的 # 字符仅仅表示 # 。代码中的注释不会被 Python 解释，录入示例的时候可以忽 略它们。

Some examples: 如下示例

```
# this is the first comment
SPAM = 1                 # and this is the second comment
                         # ... and now a third!
STRING = "# This is not a comment."
```

3.1 Using Python as a Calculator 将 Python 当做计算器

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

我们来尝试一些简单的 Python 命令。启动解释器然后等待主提示符 >>> 出现（不需要很久）。

3.1.1 Numbers 数值

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

解释器的表示就像一个简单的计算器：可以向其录入一些表达式，它会给出返回 值。表达式语法很直白：运算符 + ， - ， * 和 / 与其它语 言一样（例如： Pascal 或 C）；括号用于分组。例如

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

The equal sign ('=') is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

等号（ '=' ）用于给变量赋值

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

一个值可以同时赋给几个变量

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Variables must be ``defined'' (assigned a value) before they can be used, or an error will occur:

变量在使用前必须“定义”（赋值），否则会出错

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

浮点数有完整的支持；与整型混合计算时会自动转为浮点数

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Complex numbers are also supported; imaginary numbers are written with a suffix of j or J. Complex numbers with a nonzero real component are written as (real+imagj), or can be created with the complex(real, imag) function. :

复数也得到支持；带有后缀 j 或 J 就被视为虚数。带有非零实部的复 数写为 (real+imagj) ，或者可以用 complex(real, imag) 函数创建

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number z, use z.real and z.imag. :

复数的实部和虚部总是记为两个浮点数。要从复数 z 中提取实部和虚部，使 用 z.real 和 z.imag 。

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

The conversion functions to floating point and integer (float(), int() and long()) don't work for complex numbers --- there is no one correct way to convert a complex number to a real number. Use abs(z) to get its magnitude (as a float) or z.real to get its real part. :

浮点数和整数之间的转换函数（ float() 和 int() 以及 long() ） 不能用于复数。没有什么正确方法可以把一个复数转成一个实 数。函数 abs(z) 用于获取其模（浮点数）或 z.real 获取其实部

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

交互模式中，最近一个表达式的值赋给变量 _ 。这样我们就可以把它当作 一个桌面计算器，很方便的用于连续计算，例如

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it --- you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

此变量对于用户是只读的。不要尝试给它赋值 —— 你只会创建一个独立的同名局 部变量，它屏蔽了系统内置变量的魔术效果。

3.1.2 Strings 字符串

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes:

相比数值，Python 也提供了可以通过几种不同方式传递的字符串。它们可以用 单引号或双引号标识

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

字符串文本有几种方法分行。可以使用反斜杠为行结尾的连续字符串，它表示下 一行在逻辑上是本行的后续内容

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print hello
```

Note that newlines still need to be embedded in the string using \n -- the newline following the trailing backslash is discarded. This example would print the following:

需要注意的是，还是需要在字符串中写入 \n ——结尾的反斜杠会被忽略。前 例会打印为如下形式：

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Or, strings can be surrounded in a pair of matching triple-quotes: """ or '''. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string. :

另外，字符串可以标识在一对儿三引号中： """ 或 ''' 。三引号中， 不需要行属转义，它们已经包含在字符串中

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

produces the following output:

得到以下输出

.. code-block:: text

```
     Usage: thingy [OPTIONS]

               -h                  Display this usage message

               -H hostname         Hostname to connect to
```

If we make the string literal a ``raw'' string, \n sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

如果我们生成一个“原始”字符串， \n 序列不会被转义，而且行尾的反斜 杠，源码中的换行符，都成为字符串中的一部分数据，因此下例

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print hello
```

would print:

会打印：

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

The interpreter prints the result of string operations in the same way as they are typed for input: inside quotes, and with quotes and other funny characters escaped by backslashes, to show the precise value. The string is enclosed in double quotes if the string contains a single quote and no double quotes, else it's enclosed in single quotes. (The `print` statement, described later, can be used to write strings without quotes or escapes.)

解释器打印的字符串操作结果与它们输入时的方式一致：以括号标识，包含反斜 杠转义的有趣的字符，以精确的显示值。如果字符串包含单引号，不包含双引 号，它就以双引号标识。否则它以单引号标识。（后面介绍的 `print` 语句，可以输出没有标识和转义的字符串。）

Strings can be concatenated (glued together) with the + operator, and repeated with *:

字符串可以由 + 操作符连接（粘到一起），可以由 * 重复

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Two string literals next to each other are automatically concatenated; the first line above could also have been written `word = 'Help' 'A'`; this only works with two literals, not with arbitrary string expressions:

相邻的两个字符串文本自动连接在一起，前面那行代码也可以写为 `word ='Help' 'A'` ，它只用于两个字符串文本，不能用于字符串表达式。

```
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> 'str'.strip() + 'ing'   #  <-  This is ok
'string'
>>> 'str'.strip() 'ing'     #  <-  This is invalid
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                      ^
SyntaxError: invalid syntax
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the slice notation: two indices separated by a colon. :

字符串也可以被截取（检索）。类似于 C ，字符串的第一个字符索引为 0 。没 有独立的字符类型，字符就是长度为 1 的字符串。类似 Icon ，可以用 切片标注 法截取字符串：由两个索引分割的复本。

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced. :

索引切片可以有默认值，切片时，忽略第一个索引的话，默认为0，忽略第二个 索引，默认为字符串的长度。（其实还有第三个参数，表示切片步长，它默认为 1，完整的切片操作是 word[2:4:1] ——译者）

```
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # Everything except the first two characters
'lpA'
```

Unlike a C string, Python strings cannot be changed. Assigning to an indexed position in the string results in an error:

不同于 C 字符串，Python 字符串不可变。向字符串文本的某一个索引赋值会引 发错误

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
```

However, creating a new string with the combined content is easy and efficient:

不过，组合文本内容生成一个新文本简单而高效

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Here's a useful invariant of slice operations: `s[:i] + s[i:]` equals `s`. :

切片操作有个有用的不变性： `s[:i] + s[i:]` 等于 `s` 。

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Degenerate slice indices are handled gracefully: an index that is too large is replaced by the string size, an upper bound smaller than the lower bound returns an empty string. :

退化的切割检索很优雅：上限过大，会替换为文本长度，上界小于下界则返回空 字符串。

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Indices may be negative numbers, to start counting from the right. For example:

索引可以是负数，此时从右端开始计量。例如

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

But note that -0 is really the same as 0, so it does not count from the right! :

不过需要注意的是 -0 实际上就是 0，所以它不会从右边开始计数！

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

Out-of-range negative slice indices are truncated, but don't try this for single-element (non-slice) indices:

负索引切片越界会被截断，不要尝试将它用于单元素（非切片）检索

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n, for example:

有个办法可以很容易的记住切片的工作方式：切片时的索引是在两个字符 之间 。左边第一个字符的索引为 0，，而长度为 n 的字符串其最后一个字 符的右界索引为 n 。例如

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...5 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively.

文本中的第一行数字给出字符串中的索引点 0...5 。第二行给出相应的负索引。 切片是从 i 到 j 两个数值标示的边界之间的所有字符。

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

对于非负索引，如果上下都在边界内，切片长度就是索引值的差。例如， `word[1:3]` 是 2 。

The built-in function `len()` returns the length of a string:

内置函数 `len()` 返回字符串长度

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See Also:

typesseq Strings, and the Unicode strings described in the next section, are examples of sequence types, and support the common operations supported by such types.

string-methods Both strings and Unicode strings support a large number of methods for basic transformations and searching.

new-string-formatting Information about string formatting with `str.format()` is described here.

string-formatting The old formatting operations invoked when strings and Unicode strings are the left operand of the `%` operator are described in more detail here.

3.1.3 Unicode Strings Unicode 文本

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see http://www.unicode.org/) and integrates well with the existing string objects, providing auto-conversions where necessary.

从 Python 2.0 起，程序员们有了一个新的，用来存储文本数据的类型： Unicode 对象。它可以用于存储和维护 Unicode 数据（参见 http://www.unicode.org/ ） ，并且与现有的字符串对象有良好的集成，必要 时提供自动转换。

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as `i18n` --- `'i'` + 18 characters + `'n'`) of software. Unicode solves these problems by defining one code page for all scripts.

Unicode 的先进之处在于为每一种现代或古代使用的文字系统中出现的每一个字符都 提供了统一的序列号。之前，文字系统中的字符只能有 256 种可能的顺序。通过代 码页分界映射。文本绑定到映射文字系统

的代码页。这在软件国际化的时候尤其 麻烦 （通常写作 i18n —— 'i' + 18 个字符 + 'n' ）。 Unicode 解决了为所有的文字系统设置一个独立代码页的难题。

Creating Unicode strings in Python is just as simple as creating normal strings:

在 Python 中创建 Unicode 字符串和创建普通的字符串一样简单

```
>>> u'Hello World !'
u'Hello World !'
```

The small 'u' in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python Unicode-Escape encoding. The following example shows how:

引号前的 'u' 表示这会创建一个 Unicode 字符串。如果想要在字符串中包 含特殊字符，可以使用 Python 的 Unicode-Escape （Unicode 转义——译者）。 请看下面的例子

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with `ur' to have unevenPython use the Raw-Unicode-Escape encoding. It will only apply the above \uXXXX conversion if there is an uneven number of backslashes in front of the small `u'. :

特别的，和普通字符串一样， Unicode 字符串也有原始模式。可以在引号前加 “ur”，Python 会采用 Raw-Unicode-Escape 编码（原始 Unicode 转义——译 者）。即使有奇数个反斜杠前缀，形如 `u' ，的数值，它也只会显示为 \uXXXX 。

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

如果你需要大量输入反斜杠，原始模式非常有用，这在正则表达式中几乎是必须 的。

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

所为这些编码标准的一部分，Python 提供了基于已知编码来创建 Unicode 字符 串的整套方法。
The built-in function unicode() provides access to all registered Unicode codecs (COders and DECoders). Some of the more well known encodings which these codecs can convert are Latin-1, ASCII, UTF-8, and UTF-16. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to ASCII, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with str(), conversion takes place using this default encoding. :

内置函数 unicode() 可以使用所有注册的 Unicode 编码（ COders 和 DECoders ）。众所周知， Latin-1 ， ASCII ， UTF-8 和 UTF-16 之 类的编码可以互相转换（Latin-1 表示一个很小的拉丁语言符号集，与 ASCII 基 本一致，其实不能用来表示庞大的东方语言字符集——译者）。后两个是变长编 码，将每一个 Uniocde 字符存储为一到多个字节。默认通常编码为 ASCII，此 编码接受 0 到 127 这个范围的编码，否则报错。将一个 Unicode 字符串打印 或写入到文件中，或者使用 str() 转换时，转换操作以此为默认编码。

```
>>> u"abc"
u'abc'
>>> str(u"abc")
```

```
'abc'
>>> u""
u'\xe4\xf6\xfc'
>>> str(u"")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred. :

为了将一个 Unicode 字符串写为一个使用特定编码的 8 位字符串， Unicode 对象提供一 `encode()` 方法，它接受编码名作为参数。编码名应该小写。

```
>>> u"".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

If you have data in a specific encoding and want to produce a corresponding Unicode string from it, you can use the `unicode()` function with the encoding name as the second argument. :

如果有一个其它编码的数据，希望可以从中生成一 Unicode 字符串，你可以使用 `unicode()` 函数，它接受编码名作为第二参数。

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Lists 列表

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type. :

Python 有几个 复合 数据类型，用于分线其它的值。最通用的是 list (列 表) ，它可以写作中括号之间的一列逗号分隔的值。列表的元素不必是同一类型。

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

就像字符串索引，列表从 0 开始检索。列表可以被切片和连接

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list a:

所有的切片操作都会返回新的列表，包含求得的元素。这意味着以下的切片操作 返回列表 a 的一个浅复制副本

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are immutable, it is possible to change individual elements of a list:

不像 不可变的 字符串，列表允许修改元素

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

也可以对切片赋值，此操作可以改变列表的尺寸，或清空它

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function `len()` also applies to lists:

内置函数 `len()` 也可以用于列表

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

It is possible to nest lists (create lists containing other lists), for example:

允许嵌套列表（创建一个包含其它列表的列表），例如

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
```

```
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Note that in the last example, `p[1]` and `q` really refer to the same object! We'll come back to object semantics later.

注意最后一个例子中， `p[1]` 和 `q` 实际上指向同一个对象！ 我们会在 后面的 object semantics 中继续讨论。

3.2 First Steps Towards Programming 编程的第一步

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

当然，我们可以使用 Python 完成比二加二更复杂的任务。例如，我们可以写一 个生成 菲波那契 子序列的程序，如下所示

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

This example introduces several new features.

这个例子介绍了几个新功能。

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

  第一行包括了一个 多重赋值 ：变量 `a` 和 `b` 同时获得了新的值 0 和 1 。最后一行又使用了一次。在这个演示中，变量赋值前，右边首先完成 计算。右边的表达式从左到右计算。

- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).

  条件（这里是 `b < 10` ）为 true 时， `while` 循环执行。在 Python 中，类似于 C ，任何非零整数都是 true；0 是 false 。条件也可以 是字符串或列表，实际上可以是任何序列；所有长度不为零的

是 true ，空序 列是 false。示例中的测试是一个简单的比较。标准比较操作符与 C 相同： < （小于）， > （大于）， == （等于）， <= （小于等 于）， >= （大于等于）和 != （不等于）。

- The body of the loop is indented: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

  循环 体 是 缩进 的：缩进是 Python 是 Python 组织語句的方法。 Python （还） 不提供集成的行编辑功能，所以你要为每一个缩进行输入 TAB 或空格。实践中建议你找个文本编辑来录入复杂的 Python 程序，大多数文本 编辑器提供自动缩进。交互式录入复合语句时，必须在最后输入一个空行来标 识结束（因为解释器没办法猜测你输入的哪一行是最后一行），需要注意的是 同一个语句块中的语句块必须缩进同样数量的空白。

- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

  关键字 `print` 语句输出给定表达式的值。它控制多个表达式和字 符串输出为你想要字符串（就像我们在前面计算器的例子中那样）。字符串打 印时不用引号包围，每两个子项之间插入空间，所以你可以把格式弄得很漂 亮，像这样

  ```
  >>> i = 256*256
  >>> print 'The value of i is', i
  The value of i is 65536
  ```

  A trailing comma avoids the newline after the output:

  用一个逗号结尾就可以禁止输出换行

  ```
  >>> a, b = 0, 1
  >>> while b < 1000:
  ...     print b,
  ...     a, b = b, a+b
  ...
  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
  ```

  Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

  注意，在这里，如果最后一行没有输出完全，解释器在它打印下一个提示符前 会插入一个换行。

CHAPTER

FOUR

MORE CONTROL FLOW TOOLS 深入流程控制

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

除了前面介绍的 `while` 语句，Python 还从其它语言借鉴了一些流 程控制功能，并有所改变。

4.1 `if` Statements `if` 語句

Perhaps the most well-known statement type is the `if` statement. For example:

也许最有名的是 if 语句。例如

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `` `elif` `` is short for `else if', and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the `switch` or `case` statements found in other languages.

可能会有零到多个 elif 部分，else 是可选的。关键字 "elif" 是 " else if " 的缩写，这个可以有效避免过深的缩进。if ... elif ... elif ... 序列用 于替代其它语言中的 switch 或 case 语句。

4.2 `for` Statements `for` 语句

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

Python 中的 for 语句和 C 或 Pascal 中的略有不同。通常的循环可能会依据 一个等差数值步进过程（如 Pascal），或由用户来定义迭代步骤和中止条件（如 C ），Python 的 for 语句依据任意序列（链表或字符串）中的子项，按它们在序 列中的顺序来进行迭代。例如（没有暗指）：

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists). If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

在迭代过程中修改迭代序列不安全（只有在使用链表这样的可变序列时才会有这 样的情况）。如果你想要修改你迭代的序列（例如，复制选择项），你可以迭代 它的复本。使用切割标识就可以很方便的做到这一点

```
>>> for x in a[:]: # make a slice copy of the entire list
...    if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 The range() Function range() 函数

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates lists containing arithmetic progressions:

如果你需要一个数值序列，内置函数range()会很方便，它生成一个等差级 数链表

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; range(10) generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the `step'):

range(10) 生成了一个包含10个值的链表，它用链表的索引值填充了这个长度为 10的列表，所生成的链表中不包括范围中的结束值。也可以让range操作从另一 个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为 “步长”）

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine range() and len() as follows:

需要迭代链表索引的话，如下所示结合使 用range() 和 len()

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see Looping Techniques 循环技巧.

不过，这种场合可以方便的使用 `enumerate()` ，请参见 Looping Techniques 循环技巧 。

4.4 `break` and `continue` Statements, and `else` Clauses on Loops break 和 continue 语句, 以及 循环中的 else 子句

The `break` statement, like in C, breaks out of the smallest enclosing `for` or `while` loop.

break 语句和 C 中的类似，用于跳出最近的一级 for 或 while 循环。

The `continue` statement, also borrowed from C, continues with the next iteration of the loop.

`continue` 语句是从 C 中借鉴来的，它表示循环继续执行下一次迭代。

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

循环可以有一个 `else` 子句;它在循环迭代完整个列表（对于 `for` ）或执行条件 为 false （对于 `while` ）时执行，但循环被 `break` 中止的情况下不会执行。 以下搜索素数的示例程序演示了这个子句

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.5 pass Statements pass 语句

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

pass 语句什么也不做。它用于那些语法上必须要有什么语句，但程序什么也不 做的场合，例如

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

这通常用于创建最小结构的类

```
>>> class MyEmptyClass:
...     pass
...
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

另一方面， pass 可以在创建新代码时用来做函数或控制体的占位 符。可以让你在更抽象的级别上思考。pass 可以默默的被忽视

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6 Defining Functions 定义函数

We can create a function that writes the Fibonacci series to an arbitrary boundary:

我们可以定义一个函数以生成任意上界的菲波那契数列

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

关键字 def 引入了一个函数 定义 。在其后必须跟有函数名和包 括形式参数的圆括号。函数体语句从下一行开始，必须是缩进的。

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring. (More about docstrings can be found in the section Documentation Strings 文档字符串.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

函数体的第一行可以是一个字符串值，这个字符串是该函数的文档字符串 ，或称 docstring 。（更进一步的文 档字符串介绍可以在这一节找到 Documentation Strings 文档字符串 。）有些工具使用文 档字符串在线的生成及打印文档，或者允许用户在代码中交互式的浏览；编写代 码进加入文档字符串是个好的风格，应该养成习惯。

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

执行 函数时会为局部变量引入一个新的符号表。所有的局部变量都存储在这 个局部符号表中。引用参数时，会先从局部符号表中查找，然后是全局符号表，然 后是内置命名表。因此，全局参数虽然可以被引用，但它们不能在函数中直接 赋值（除非它们用 `global` 语句命名）。

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object). [1] When a function calls another function, a new local symbol table is created for that call.

函数引用的实际参数在函数调用时引入局部符号表，因此，实参总是 传值调用 （这里的 值 总是一个对象引用 ，而不是该对象的值）。[2] 一个函数被另一个函 数调用时，一个新的局部符号表在调用过程中被创建。

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

函数定义在当前符号表中引入函数名。作为用户定义函数，函数名有一个为解释 器认可的类型值。这个值可以赋给其它命名，使其能够作为一个函数来使用。这 就像一个重命名机制

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print`:

你可能认为 `fib` 没有返回值，它不是一个函数（ function ），而是一个 过程（ procedure ）。实际上，即使函数没有 `return` 语句，它也 有返回值，虽然是一个不讨人喜欢的。这个值被称为 `None` （这是一个内置命 名）。如果一个值只是 None 的话，通常解释器不会写出来，如果你真想要查看 它的话，可以这样做

```
>>> fib(0)
>>> print fib(0)
None
```

[1] Actually, call by object reference would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

[2] 实际上， 引用对象调用 描述的更为准确。如果传入一个可变对像，调用 者会看到调用操作带来的任何变化（如子项插入到列表中）。

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

以下示例演示了如何从函数中返回一个包含菲波那契数列的数值链表，而不是打 印它

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

和以前一样，这个例子演示了一些新的 Python 功能：

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.

  `return` 语句从函数中返回一个值，不带表达式的 `return` 返回 `None` 。过程结束后也会返回 `None` 。

- The statement `result.append(a)` calls a method of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using classes, see Classes 类) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

  语句 `result.append(b)` 称为链表对象 `result` 的一个 方法 (method) 。方 法是一个"属于"某个对象的函数，它被命名为 `obj.methodename` ，这里的 `obj` 是某个对象（可能是一个表达式），`methodename` 是某个在该对象类型定 义中的方法的命名。不同的类型定义不同的方法。不同类型可能有同样名字的 方法，但不会混淆。（当你定义自己的对象类型和方法时，可能会出现这种情 况，class 的定义方法详见 Classes 类 ）。示例中演示的 `append()` 方法 由链表对象定义，它向链表中加入一个新元素。在示例中它等同于

`result = result + [b]` ，不过效率更高。

4.7 More on Defining Functions 深入函数定义

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

有时需要定义参数个数可变的函数。有三种形式，我们可以组合使用它们。

4.7.1 Default Argument Values 参数默认值

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

最有用的形式是给一个或多个参数指定默认值。这样创建的函数可以用较少的参 数来调用。例如

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

This function can be called in several ways:

这个函数可以通过几种不同的方式调用 :

- giving only the mandatory argument: 只给出必要的参数 : `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: 给出一个可选的参数 : `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: 或者给出所有的参数 : `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

这个例子还介绍了 `in` 关键字。它测定序列中是否包含某个确定的 值。

The default values are evaluated at the point of function definition in the defining scope, so that :

默认值在函数 定义 作用域被解析，如下所示

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

will print `5`.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls :

重要警告 默认值只解析一次。这造成字典、列表或大部分类实例等可变对象的行为会与期待的不太一样。例如，下例的函数在每次调用时都造成参数的累加

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

This will print :

这样会打印出

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

如果你不想在随后的调用中共享默认值，可以像这样写函数

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Keyword Arguments 关键字参数

Functions can also be called using keyword arguments of the form `keyword = value`. For instance, the following function:

函数可以通过关键字参数的形式来调用，形如 `keyword = value` 。例如， 以下的函数

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

could be called in any of the following ways:

可以用以下的任一方法调用

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

but the following calls would all be invalid:

不过以下几种调用是无效的

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument following keyword
parrot(110, voltage=220)     # duplicate value for argument
parrot(actor='John Cleese')  # unknown keyword
```

In general, an argument list must have any positional arguments followed by any keyword arguments, where the keywords must be chosen from the formal parameter names. It's not important whether a formal parameter has a default value or not. No argument may receive a value more than once --- formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls. Here's an example that fails due to this restriction:

通常，参数列表中的每一个关键字都必须来自于形式参数，每个参数都有对应的 关键字。形式参数有没有默认值并不重要。实际参数不能一次赋多个值——形式参 数不能在同一次调用中同时使用位置和关键字绑定值。这里有一个例子演示了在 这种约束下所出现的失败情况

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form **name is present, it receives a dictionary (see typesmapping) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form *name (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.) For example, if we define a function like this:

引入一个形如 **name 的参数时，它接收一个字典（参见 typesmapping ） ，该字典包含了所有未出现在形式参数列表中的关键字参数。这里可能还会组合使用一个形如 *name （下一小节詳細介绍） 的形 式参数，它接收一个元组（下一节中会详细介绍），包含了所有没有出现在形式 参数列表中的参数值。（ *name 必须在 **name 之前出现） 例如，我们这样定 义一个函数

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print "-" * 40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ":", keywords[kw]
```

It could be called like this:

它可以像这样调用

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

当然它会按如下内容打印

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the sort() method of the list of keyword argument names is called before printing the contents of the keywords dictionary; if this is not done, the order in which the arguments are printed is undefined.

注意在打印 参数字典的内容前先调用 sort() 方法。否则的话，打印参数时的顺序是未定义的。

4.7.3 Arbitrary Argument Lists 可变参数列表

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see Tuples and Sequences 元组和序列). Before the variable number of arguments, zero or more normal arguments may occur. :

最后，一个最不常用的选择是可以让函数调用可变个数的参数。这些参数被包装 进一个元组（参见 Tuples and Sequences 元组和序列 ）。在这些可变个数的参数之前，可以有零到多个普通的参数

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

4.7.4 Unpacking Argument Lists 参数列表的分拆

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate start and stop arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

另有一种相反的情况: 当你要传递的参数已经是一个列表，但要调用的函数却接受 分开一个个的参数值. 这时候你要把已有的列表拆开来. 例如内建函数 `range()` 需要要独立的 start, stop 参数. 你可以在调用函数时加一个 `*` 操作符来自动 把参数列表拆开

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

以同样的方式，可以使用 `**` 操作符分拆关键字参数为字典

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

4.7.5 Lambda Forms Lambda 形式

By popular demand, a few features commonly found in functional programming languages like Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: `lambda a, b: a+b`. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

出于实际需要，有几种通常在函数式编程语言例如 Lisp 中出现的功能加入到了 Python 。通过 `lambda` 关键字，可以创建短小的匿名函数。这里有一个函数返 回它的两个参数的和： `lambda a, b: a+b` 。

Lambda 形式可以用于任何需要的 函数对象。出于语法限制，它们只能有一个单独的表达式。语义上讲，它们只是 普通函数定义中的一个语法技巧。类似于嵌套函数定义，lambda 形式可以从外 部作用域引用变量

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.6 Documentation Strings 文档字符串

There are emerging conventions about the content and formatting of documentation strings.

这里介绍的文档字符串的概念和格式。

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

第一行应该是关于对象用途的简介。简短起见，不用明确的陈述对象名或类型， 因为它们可以从别的途径了解到（除非这个名字碰巧就是描述这个函数操作的动 词）。这一行应该以大写字母开头，以句号结尾。

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

如果文档字符串有多行，第二行应该空出来，与接下来的详细描述明确分隔。接 下来的文档应该有一或多段描述对象的调用约定、边界效应等。

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace ``equivalent'' to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Python 的解释器不会从多行的文档字符串中去除缩进，所以必要的时候应当自己 清除缩进。这符合通常的习惯。第一行之后的第一个非空行决定了整个文档的缩 进格式。（我们不用第一行是因为它通常紧靠着起始的引号，缩进格式显示的不 清楚。）留白“相当于”是字符串的起始缩进。每一行都不应该有缩进，如果有 缩进的话，所有的留白都应该清除掉。留白的长度应当等于扩展制表符的宽度 （通常是8个空格）。

Here is an example of a multi-line docstring:

以下是一个多行文档字符串的示例

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
```

```
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8 Intermezzo: Coding Style 插曲：编码风格

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about coding style. Most languages can be written (or more concise, formatted) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

此时你已经可以写一此更长更复杂的 Python 程序，是时候讨论一下 编码风格 了。大多数语言可以写（或者更明白的说， 格式化 ）作几种不 同的风格。有些比其它的更好读。让你的代码对别人更易读是个好想法，养成良 好的编码风格对此很有帮助。

For Python, PEP 8 has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

对于 Python， PEP 8 引入了大多数项目遵循的风格指导。它给出了一个高 度可读，视觉友好的编码风格。每个 Python 开发者都应该读一下，大多数要点 都会对你有帮助：

- Use 4-space indentation, and no tabs.

  使用 4 空格缩进，而非 TAB。

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

  在小缩进（可以嵌套更深）和大缩进（更易读）之间，4空格是一个很好的折 中。TAB 引发了一些混乱，最好弃用。

- Wrap lines so that they don't exceed 79 characters.

  折行以确保其不会超过 79 个字符。

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

  这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件。

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

  使用空行分隔函数和类，以及函数中的大块代码。

- When possible, put comments on a line of their own.

  可能的话，注释独占一行

- Use docstrings.

  使用文档字符串

- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.

  把空格放到操作符两边，以及逗号后面，但是括号里侧不加空格： `a = f(1, 2) + g(3, 4)` 。

- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see A First Look at Classes 初识类 for more on classes and methods).

  统一函数和类命名。推荐类名用 `` `` _ _ 。 总是用 `self` 作为方法的第一个参数（关于类和 方法的知识详见 A First Look at Classes 初识类 ）。

- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

在国际化环境中不要随意使用自己喜欢的编码， 纯 ASCII 文本总是工作的最好。（作为东方人，我有不同的见解，个人推荐首选 utf-8——译者）

CHAPTER

FIVE

DATA STRUCTURES 数据结构

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

本章会深入讨论你一些之前已经学过的，而有些是新内容。

5.1 More on Lists 深入列表

The list data type has some more methods. Here are all of the methods of list objects:

链表类型有很多方法，这里是链表类型的所有方法：

list.**append**(x)
Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

把一个元素添加到链表的结尾，相当于 `a[len(a):] = [x]` 。

list.**extend**(L)
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

将一个给定列表中的所有元素都添加到另一个列表中，相当于 `a[len(a):] = L` 。

list.**insert**(i, x)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 `a.insert(0, x)` 会插入到整个链表之前，而 `a.insert(len(a), x)` 相当于 `a.append(x)` 。

list.**remove**(x)
Remove the first item from the list whose value is x. It is an error if there is no such item.

删除链表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。

list.**pop**([i])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

从链表的指定位置删除元素，并将其返回。如果没有指定索引， a.pop() 返 回最后一个元素。元素随即从链表中被删除。（方法中 i 两边的方括号表示 这个参数是可选的，而不是要求你输入一对方括号，你会经常在Python 库参 考手册中遇到这样的标记。）

list.index(x)
Return the index in the list of the first item whose value is x. It is an error if there is no such item.

返回链表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。

list.count(x)
Return the number of times x appears in the list. 返回 x 在链表中出现的次数。

list.sort()
Sort the items of the list, in place.

对链表中的元素就地（原文 in place，意即该操作直接修改调用它的对象——译者）进行排序。

list.reverse()
Reverse the elements of the list, in place.

就地倒排链表中的元素。

An example that uses most of the list methods:

下面这个示例演示了链表的大部分方法

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

5.1.1 Using Lists as Stacks 把链表当作堆栈使用

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (``last-in, first-out''). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

链表方法使得链表可以很方便的做为一个堆栈来使用，堆栈作为特定的数据结 构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一 个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放 出来。例如

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Using Lists as Queues 把链表当作队列使用

It is also possible to use a list as a queue, where the first element added is the first element retrieved (``first-in, first-out''); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

你也可以把链表当做队列使用，队列作为特定的数据结构，最先进入的元素最先 释放（先进先出）。不过，列表这样用效率不高。相对来说从列表末尾添加和弹 出很快；在头部插入和弹出很慢（因为，为了一个元素，要移动整个列表中的所 有元素）。

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

要实现队列，使用 `collections.deque` ，它为在首尾两端快速插入和 删除而设计。例如

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Functional Programming Tools 函数式编程工具

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

对于链表来讲，有三个内置函数非常有用： `filter()` ， `map`` :func:`reduce()` 。

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If sequence is a `string` or `tuple`, the result will be of the same type; otherwise, it is always a `list`. For example, to compute some primes:

`filter(function, sequence)` 返回一个sequence（序列），包括了给定序 列中所有调用 `function(item)` 后返回值为true的元素。（如果可能的话， 会返回相同的类型）。如果该 序列（sequence） 是一个 `string` （字符串）或者 `tuple` （元组），返回值必定是同一类型，否则，它 总是 `list` 。例如，以下程序可以计算部分素数

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

`map(function, sequence)` 为每一个元素依次调用 `function(item)` 并将返回值 组成一个链表返回。例如，以下程序计算立方

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). For example:

可以传入多个序列，函数也必须要有对应数量的参数，执行时会依次用各序列上 对应的元素来调用函数（如果某些序列比其它的短，就用 `None` 来代替）。如果把 None做为一个函数传入，则直接返回参数做为替代。例如

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function function on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

`reduce(func, sequence)` 返回一个单值，它是这样构造的：首先以序列的 前两个元素调用函数 function ，再以返回值和第三个参数调用，依次执行下去。例如，以 下程序计算 1 到 10 的整数之和

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

如果序列中只有一个元素，就返回它，如果序列是空的，就抛出一个异常。

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example, :

可以传入第三个参数做为初始值。如果序列是空的，就返回初始值，否则函数会 先接收初始值和序列的第一个元素，然后是返回值和下一个元素，依此类推。例 如

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
```

```
>>> sum([])
0
```

Don't use this example's definition of sum(): since summing numbers is such a common need, a built-in function sum(sequence) is already provided, and works exactly like this.

不要像示例中这样定义 sum() ：因为合计数值是一个通用的需求，早已 有内置的 sum(sequence) 函数，非常好用。 New in version 2.3.

5.1.4 List Comprehensions 列表推导式

List comprehensions provide a concise way to create lists without resorting to use of map(), filter() and/or lambda. The resulting list definition tends often to be clearer than lists built using those constructs. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the for and if clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized. :

列表推导式提供了一个创建链表的简单途径，无需使用 map() ， filter() 以及 lambda 。以定义方式得到列表通常要比使用构造函数创建这些列表更清晰。每一个列表推导式包括 在一个 for 语句之后的表达式，零或多个 for 或 if 语句。返回值是由 for 或 if 子句之后的表达式得到的元素组成的列表。如果想要得到一个元组，必须要加 上括号。

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

List comprehensions are much more flexible than map() and can be applied to complex expressions and nested functions:

列表推导式比 map() 更复杂，可使用复杂的表达式和嵌套函数

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.5 Nested List Comprehensions 嵌套的列表推导式

If you've got the stomach for it, list comprehensions can be nested. They are a powerful tool but -- like all powerful tools -- they need to be used carefully, if at all.

如果你不会晕菜的话，列表推导式可以嵌套。它们可以是非常强力的工具——就像 所有的强力工具一样——它们用起来也需要格外小心。

Consider the following example of a 3x3 matrix held as a list containing three lists, one list per row:

考虑以下的 3x3 矩阵的例子，一个列表中包含了三个列表，每个一行

```
>>> mat = [
...         [1, 2, 3],
...         [4, 5, 6],
...         [7, 8, 9],
...       ]
```

Now, if you wanted to swap rows and columns, you could use a list comprehension:

现在，如果你想交换行和列，可以用列表推导式

```
>>> print [[row[i] for row in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Special care has to be taken for the nested list comprehension:

嵌套 列表推导式的时候要特别小心：

> To avoid apprehension when nesting list comprehensions, read from right to left.
>
> 为了不被嵌套的列表推导式搞晕，从右往左读。

A more verbose version of this snippet shows the flow explicitly:

接下来有一个更容易读的版本

```
for i in [0, 1, 2]:
    for row in mat:
        print row[i],
    print
```

In real world, you should prefer built-in functions to complex flow statements. The zip() function would do a great job for this use case:

实用中，你可以利用内置函数完成复杂的流程语句。函数 zip() 在这个 例子中可以搞定大量的工作

```
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

See Unpacking Argument Lists 参数列表的分拆 for details on the asterisk in this line.

关于这行代码中带有星号的参数，请参见 Unpacking Argument Lists 参数列表的分拆 。

5.2 The del statement 删除语句

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

有个方法可以从列表中按给定的索引而不是值来删除一个子项： del 语句。它不同于有返回值的 pop() 方法。语句 del 还可以从列表中删除切片或清空整个列表（我们以前介绍过一 个方法是将空列表赋值给列表的切片）。例如

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del can also be used to delete entire variables:

del 也可以删除整个变量

```
>>> del a
```

Referencing the name a hereafter is an error (at least until another value is assigned to it). We'll find other uses for del later.

此后再引用命名 a 会引发错误（直到另一个值赋给它为止）。我们在后面 的内容中可以看到 del 的其它用法。

5.3 Tuples and Sequences 元组和序列

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of sequence data types (see typesseq). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the tuple.

我们知道链表和字符串有很多通用的属性，例如索引和切割操作。它们是 序 列 类型（参见 typesseq ）中的两种。因为 Python 是一个在不停进化的语言，也可能会加入其它的序列类型，这里介绍另一种标准序列类型： 元组 。

A tuple consists of a number of values separated by commas, for instance:

一个元组由数个逗号分隔的值组成，例如

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可 以有或没有括号，不过经常括号都是必须的（如果元组是一个更大的表达 式的一部分）。

Tuples have many uses. For example: (x, y) coordinate pairs, employee records from a database, etc. Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though). It is also possible to create tuples which contain mutable objects, such as lists.

元组有很多用途。例如 (x, y) 坐标对，数据库中的员工记录等等。元组就像字符 串，不可改变：不能给元组的一个独立的元素赋值（尽管你可以通过联接和切割 来模拟）。还可以创建包含可变对象的元组，例如链表。

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

一个特殊的问题是构造包含零个或一个元素的元组：为了适应这种情况，语法上 有一些额外的改变。一对空的括号可以创建空元组；要创建一个单元素元组可以 在值后面跟一个逗号（在括号中放入一个单值不够明确）。丑陋，但是有效。例 如

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

语句 `t = 12345, 54321, 'hello!'` 是 元组封装 （tuple packing）的 一个例子：值 `12345` ， `54321` 和 `'hello!'` 被封装进元组。其逆 操作可能是这样

```
>>> x, y, z = t
```

This is called, appropriately enough, sequence unpacking and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

这个调用等号右边可以是任何线性序列，称之为 序列拆封 非常恰当。序列拆 封要求左侧的变量数目与序列的元素个数相同。要注意的是可变参数（multiple assignment ）其实只是元组封装和序列拆封的一个结合。

5.4 Sets 集合

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Python 还包含了一个数据类型—— set（集合） 。集合是一个无序不重复元素的 集。基本功能包括关系测试和消除重复元素。集合对象还支持 union（联 合），intersection（交），difference（差）和 sysmmetric difference（对 称差集）等数学运算。

Here is a brief demonstration:

以下是一个简单的演示

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                              # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # letters in both a and b
set(['a', 'c'])
>>> a ^ b                              # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.5 Dictionaries 字典

Another useful data type built into Python is the dictionary (see typesmapping). Dictionaries are sometimes found in other languages as ``associative memories'' or ``associative arrays''. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

另一个非常有用的 Python 内建数据类型是 字典 （参见 typesmapping ）。字典在某些语言中可能称为 （ associative memories ）或 （ associative arrays ）。序列是以连续的整数为索引，与此不同的是，字 典以 关键字 为索引，关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字 符串和数字，它可以做为关键字，如果它直接或间接的包含了可变对象，就不能 当做关键字。不能用链表做关键字，因为链表可以用索引、切割或者 append() 和 extend() 等方法改变。

It is best to think of a dictionary as an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

理解字典的最佳方式是把它看做无序的键：值 对（key:value pairs）集 合，键必须是互不相同的（在同一个字典之内）。一对大括号创建一个空的 字典： {} 。初始化链表时，在大括号内放置一组逗号分隔的键：值对，这也 是字典输出的方式。

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

字典的主要操作是依据键来存储和析取值。也可以用 `del` 来删除键： 值对（key:value）。如果你用一个已经存在的关键字存储值，以前为该关键字 分配的值就会被遗忘。试图从一个不存在的键中取值会导致错误。

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sort()` method to the list of keys). To check whether a single key is in the dictionary, use the `in` keyword.

字典的 `keys()` 方法返回由所有关键字组成的链表，该链表的顺序不定（如果你 需要它有序，只能调用关键字链表的 `sort()` 方法）。可以用 `in` 关键字检查字典中是否存在某一关键字。

Here is a small example using a dictionary:

这里有个字典用法的小例子

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

The `dict()` constructor builds dictionaries directly from lists of key-value pairs stored as tuples. When the pairs form a pattern, list comprehensions can compactly specify the key-value list. :

链表中存储关键字-值对元组的话，:func:dict 可以从中直接构造字典。键-值 对来自某个特定模式时，可以用链表推导式简单的生成关键字-值链表。

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Later in the tutorial, we will learn about Generator Expressions which are even better suited for the task of supplying key-values pairs to the `dict()` constructor.

在入门指南后面的内容中，我们将会学习更适于为 `dict()` 构造器生成键值对的生成器表达式。

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

使用简单字符串作为关键字的话，通常用关键字参数更简单

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6 Looping Techniques 循环技巧

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method. :

在字典中循环时，关键字和对应的值可以使用 `iteritems()` 方法同时解读出来

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function. :

在序列中循环时，索引位置和对应值可以使用 `enumerate()` 函数同时得 到。

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function. :

同时循环两个或更多的序列，可以使用 zip() 整体打包。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function. :

需要逆向循环序列的话，先正向定位序列，然后调用 `reversed()` 函数

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

要按排序后的顺序循环序列的话，使用 `sorted()` 函数，它不改动原序列，而是 生成一个新的已排序的序列。

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

5.7 More on Conditions 深入条件控制

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

`while` 和 `if` 语句中使用的条件不仅可以使用比较，而且可以包含任意的操作。

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

比较操作符 `in` 和 `not in` 审核值是否在一个区间之内。操作符 `is` 和 `is not` 比较两个对象是否相同；这只和诸如链表这样的可变对 象有关。所有的比较操作符具有相同的优先级，低于所有的数值操作。

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

比较操作可以传递。例如 `a < b == c` 审核是否 `a` 小于 `b` 并且 `b` 等于 `c` 。

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

比较操作可以通过逻辑操作符 `and` 和 `or` 组合，比较的结果可以用 `not` 来取反 义。这些操作符的优先级又低于比较操作符，在它们之中，``not`` 具有最高的优先 级， `or` 优先级最低，所以 `A and not B or C` 等于 `(A and (notB)) or C` 。当然，括号也可以用于比较表达式。

The Boolean operators `and` and `or` are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

逻辑操作符 `and` 和 `or` 也称作 短路操作符 ：它们的参数从左向右解 析，一旦结果可以确定就停止。例如，如果 `A` 和 `C` 为眞而 `B` 为 假， `A and B and C` 不会解析 `C`。作用于一个普通的非逻辑值时，短路操作 符的返回值通常是最后一个变量。

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example, :

可以把比较或其它逻辑表达式的返回值赋给一个变量，例如

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

需要注意的是Python与C不同，在表达式内部不能赋值。C 程序员经常对此抱怨，不 过它避免了一类在 C 程序中司空见惯的错误：想要在解析式中使 == 时误用了 = 操作符。

5.8 Comparing Sequences and Other Types 比较序列和其它类型

Sequence objects may be compared to other objects with the same sequence type. The comparison uses lexicographical ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

序列对象可以与相同类型的其它对象比较。比较操作按 字典序 进行：首先比较 前两个元素，如果不同，就决定了比较的结果；如果相同，就比较后两个元素， 依此类推，直到所有序列都完成比较。如果两个元素本身就是同样类型的序列， 就递归字典序比较。如果两个序列的所有子项都相等，就认为序列相等。如果一 个序列是另一个序列的初始子序列，较短的一个序列就小于另一个。字符串的字 典序按照单字符的 ASCII 顺序。下面是同类型序列之间比较的一些例子

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc. [1] Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

需要注意的是不同类型的对象比较是合法的。输出结果是确定而非任意的：类型 按它们的名字排序。因而，一个链表（list）总是小于一个字符串（string）， 一个字符串（string）总是小于一个元组（tuple）等等。[2] 数值类型比较时会统 一它们的数据类型，所以0等于0.0，等等。

[1] The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.

[2] Python 并不承诺不同类型之间进行比较时的明确规则，当前的方式在未来的 版本中可能会改变

CHAPTER

SIX

MODULES 模块

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

如果你退出 Python 解释器重新进入，以前创建的一切定义（变量和函数）就全 部丢失了。因此，如果你想写一些长久保存的程序，最好使用一个文本编辑器来 编写程序，把保存好的文件输入解释器。我们称之为创建一个 脚本 。或者程 序变得更长了，你可能为了方便维护而把它分离成几个文件。你也可能想要在几 个程序中都使用一个常用的函数，但是不想把它的定义复制到每一个程序里。

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

为了满足这些需要，Python提供了一个方法可以从文件中获取定义，在脚本或者 解释器的一个交互式实例中使用。这样的文件被称为 模块 ；模块中的定义可 以 导入 到另一个模块或主模块中（在脚本执行时可以调用的变量集位于最高级，并且处 于计算器模式）。

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

模块是包括 Python 定义和声明的文件。文件名就是模块名加上 .py 后缀。模 块的模块名（做为一个字符串）可以由全局变量 `__name__` 得到。例如，你可以 用自己惯用的文件编辑器在当前目录下创建一个叫 `fibo.py` 的文件，录入如下 内容

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

现在进入Python解释器，用如下命令导入这个模块

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

这样做不会直接把 `fibo` 中的函数导入当前的语义表；它只是引入了模块名 `fibo` 。你可以通过模块名按如下方式访问这个函数

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

如果你想要直接调用函数，通常可以给它赋一个本地名称

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 More on Modules 深入模块

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module is imported somewhere. [1]

模块可以像函数定义一样包含执行语句。这些语句通常用于初始化模块。它们只 在模块 第一次 导入时执行一次。 [2]

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

对应于定义模块中所有函数的全局语义表，每一个模块有自己的私有语义表。因 此，模块作者可以在模块中使用一些全局变量，不会因为与用户的全局变量冲突 而引发错误。另一方面，如果你确定你需要这个，可以像引用模块中的函数一样获取模块中的全局变量，形如： `modname.itemname` 。

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

[1] For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter -- or, if it's just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

[2] 出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修 改了你的模块，需要重启解释器——或者，如果你就是想交互式的测试这么一 个模块，可以用 `reload()` 重新加载，例如 `reload(modulename)` 。

模块可以导入（ import ）其它模块。习惯上所有的 import 语句都放在模块（或 脚本，等等）的开头，但这并不是必须的。被导入的模块名入在本模块的全局语 义表中。

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

import 语句的一个变体直接从被导入的模块中导入命名到本模块的语义表中。 例如

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

这样不会从局域语义表中导入模块名（如上所示， fibo 没有定义）。

There is even a variant to import all names that a module defines:

甚至有种方式可以导入模块中的所有定义

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_).

这样可以导入所有除了以下划线(_)开头的命名。

Note that in general the practice of importing * from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

需要注意的是在实践中往往不鼓励从一个模块或包中使用 * 导入所有，因 为这样会让代码变得很难读。不过，在交互式会话中这样用很方便省力。

6.1.1 Executing modules as scripts 作为脚本来执行模块

When you run a Python module with :

使用如下方式执行一个 Python 模块

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__". That means that by adding this code at the end of your module:

模块中的代码会被执行，就像导入它一样，不过此时 __name__ 被设置为 "__main__" 。这相当于，如果你在模块后加入如下代码

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the ``main'' file:

就可以让此文件像作为模块导入时一样作为脚本执行。此代码只有在模块作为 “main” 文件执行时才被调用

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

如果模块被导入，不会执行这段代码

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

这通常用来为模块提供一个便于测试的用户接口（将模块作为脚本执行测试需求）。

6.1.2 The Module Search Path 模块搜索路径

When a module named spam is imported, the interpreter searches for a file named spam.py in the current directory, and then in the list of directories specified by the environment variable PYTHONPATH. This has the same syntax as the shell variable PATH, that is, a list of directory names. When PYTHONPATH is not set, or when the file is not found there, the search continues in an installation-dependent default path; on Unix, this is usually .:/usr/local/lib/python.

导入一个叫 spam 的模块时，解释器先在当前目录中搜索名为 spam.py 的文件，然后在环境变量 PYTHONPATH 表示的目录 列表中搜索，然后是环境变量 PATH 中的路径列表。如果 PYTHONPATH 没有设置，或者文件没有找到，接下来搜索安装目录， 在 Unix 中，通常是 .:/usr/local/lib/python 。

Actually, modules are searched in the list of directories given by the variable sys.path which is initialized from the directory containing the input script (or the current directory), PYTHONPATH and the installation- dependent default. This allows Python programs that know what they're doing to modify or replace the module search path. Note that because the directory containing the script being run is on the search path, it is important that the script not have the same name as a standard module, or Python will attempt to load the script as a module when that module is imported. This will generally be an error. See section Standard Modules 标准模块 for more information.

实际上，解释器由 sys.path 变量指定的路径目录搜索模块，该变量初始化 时默认包含了输入脚本（或者当前目录）， PYTHONPATH 和安装目录。 这样就允许 Python 程序了解如何修改或替换模块搜索目录。需要注意的是由于 这些目录中包含有搜索路径中运行的脚本，所以这些脚本不应该和标准模块重 名，否则在导入模块时 Python 会尝试把这些脚本当作模块来加载。这通常会引 发错误。请参见 Standard Modules 标准模块 以了解更多的信息。

6.1.3 ``Compiled'' Python files “编译的” Python 文件

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called spam.pyc exists in the directory where spam.py is found, this is assumed to contain an already-``byte-compiled'' version of the module spam. The modification time of the version of spam.py used to create spam.pyc is recorded in spam.pyc, and the .pyc file is ignored if these don't match.

对于引用了大量标准模块的短程序，有一个提高启动速度的重要方法，如果在 spam.py 所在的目录下存在一个名为 spam.pyc 的文件，它会 被视为 spam 模块的预“编译”（``byte-compiled'' ，二进制编译） 版本。用于创建 spam.pyc 的这一版 spam.py 的修改时间记 录在 spam.pyc 文件中，如果两者不匹配，:file:.pyc 文件就被忽 略。

Normally, you don't need to do anything to create the spam.pyc file. Whenever spam.py is successfully compiled, an attempt is made to write the compiled version to spam.pyc. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting spam.pyc file will be recognized as invalid and thus ignored later. The contents of the spam.pyc file are platform independent, so a Python module directory can be shared by machines of different architectures.

通常你不需要为创建 spam.pyc 文件做任何工作。一旦 spam.py 成功编译，就 会尝试生成对应版本的 spam.pyc 。如果有任何原因导致写入不成功， 生成的 spam.pyc 文件就会视为无效，随后即被忽略。 spam.pyc 文件的内容是平台独 立的，所以Python模块目录可以在不同架构的机器之间共享。

Some tips for experts: 部分高级技巧：

- When the Python interpreter is invoked with the *-O* flag, optimized code is generated and stored in .pyo files. The optimizer currently doesn't help much; it only removes assert statements. When *-O* is used, all bytecode is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.

  以 *-O* 参数调用Python解释器时，会生成优化代码并保存在 .pyo 文件中。现 在的优化器没有太多帮助；它只是删除了断言（ assert ）语句。使用 *-O* 参 参数， 所有 的字节码（ bytecode ）都会被优化； .pyc 文 件被忽略， .py 文件被编译为优化代码。

- Passing two *-O* flags to the Python interpreter (*-OO*) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only __doc__ strings are removed from the bytecode, resulting in more compact .pyo files. Since some programs may rely on having these available, you should only use this option if you know what you're doing.

  向Python解释器传递两个 *-O* 参数（ *-OO* ）会执行完全 优化的二进制优化编译，这偶尔会生成错误的程序。现在的优化器，只是从字 节码中删除了 __doc__ 符串，生成更为紧凑的 .pyo 文件。因为某些程序依赖于这些变量的可用性，你应该只在确定无误的场合使用这一选项。

- A program doesn't run any faster when it is read from a .pyc or .pyo file than when it is read from a .py file; the only thing that's faster about .pyc or .pyo files is the speed with which they are loaded.

  来自 .pyc 文件或 .pyo 文件中的程序不会比来自 .py 文件的运行更快； .pyc 或 .pyo 文件只是在 它们加载的时候更快一些。

- When a script is run by giving its name on the command line, the bytecode for the script is never written to a .pyc or .pyo file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a .pyc or .pyo file directly on the command line.

  通过脚本名在命令行运行脚本时，不会将为该脚本创建的二进制代码写入 .pyc 或 .pyo 文件。当然，把脚本的主要代码移进一个模 块里，然后用一个小的启动脚本导入这个模块，就可以提高脚本的启动速度。 也可以直接在命令行中指定一个 .pyc 或 .pyo 文件。

- It is possible to have a file called spam.pyc (or spam.pyo when *-O* is used) without a file spam.py for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.

  对于同一个模块（这里指例程 spam.py ——译者），可以只 有 spam.pyc 文件（或者 spam.pyc ，在使用 *-O* 参数时）而没有 spam.py 文件。这样可以打包发布比 较难于逆向工程的 Python 代码库。

- The module compileall can create .pyc files (or .pyo files when *-O* is used) for all modules in a directory.

  compileall 模块 可以为指定目录中的所有模块创建 .pyc 文 件（或者使用 .pyo 参数创建 .pyo 文件）。

6.2 Standard Modules 标准模块

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (``Library Reference'' hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform For example, the winreg module is only provided on Windows systems. One particular module deserves some attention: sys, which is built into every Python interpreter. The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:

Python带有一个标准模块库，并发布有独立的文档，名为 Python 库参考手册 （此后称其为“库参考手册”）。有一些模块内置于解释器之中，这些操作的访 问接口不是语言内核的一部分，但是已经内置于解释器了。这既是为了提高效 率，也是为了给系统调用等操作系统原生访问提供接口。这类模块集合是一个依 赖于底层平台的配置选项。例如，:mod:winreg 模块只提供在 Windows 系统 上才有。有一个具体的模块值得注意： sys ，这个模块内置于所有的 Python 解释器。变量 sys.psl 和 sys.ps2定义了主提示符和副助提示符字符串

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

这两个变量只在解释器的交互模式下有意义。

The variable sys.path is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations:

变量 sys.path 是解释器模块搜索路径的字符串列表。它由环境变 量:envvar:PYTHONPATH 初始化，如果没有设定 PYTHONPATH ，就由 内置的默认值初始化。你可以用标准的字符串操作修改它

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 The **dir()** Function **dir()** 函数

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

内置函数 dir() 用于按模块名搜索模块定义，它返回一个字符串类型的存储列 表

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Without arguments, dir() lists the names you have defined currently:

无参数调用时， dir() 函数返回当前定义的命名

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

注意该列表列出了所有类型的名称：变量，模块，函数，等等。 dir() does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module __builtin__:

dir() 不会列出内置函数和变量名。如果你想列出这些内容，它们在标准模块 __builtin__ 中定义

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UserWarning', 'ValueError', 'Warning', 'WindowsError',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview',
 'min', 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range',
 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set',
 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Packages 包

Packages are a way of structuring Python's module namespace by using ``dotted module names''. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

包通常是使用用“圆点模块名”的结构化模块命名空间。例如，名为 `A.B` 的模块表示了名为 `B` 的包中名为 `A` 的子模块。正如同用 模块来保存不同的模块架构可以避免全局变量之间的相互冲突，使用圆点模块名 保存像 NumPy 或 Python Imaging Library 之类的不同类库架构可以避免模块 之间的命名冲突。

Suppose you want to design a collection of modules (a ``package'') for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

假设你现在想要设计一个模块集（一个“包”）来统一处理声音文件和声音数据。 存在几种不同的声音格式（通常由它们的扩展名来标识，例如： `.wav` ， `.aiff` ， `.au` ），于是，为了在不同类型的文件格式之间 转换，你需要维护一个不断增长的包集合。可能你还想要对声音数据做很多不同 的操作（例如混音，添加回声，应用平衡功能，创建一个人造效果），所以你要 加入一个无限流模块来执行这些操作。你的包可能会是这个样子（通过分级的文 件体系来进行分组）

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

导入模块时，Python通过 `sys.path` 中的目录列表来搜索存放包的子目录。

The `__init__.py` files are required to make Python treat the directories as containing packages;

this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable, described later.

必须要有一个 __init__.py 文件的存在，才能使 Python 视该目录为 一个包；这是为了防止某些目录使用了 string 这样的通用名而无意中在随 后的模块搜索路径中覆盖了正确的模块。最简单的情况下，__init__.py 可以只是一个空文件，不过它也可能包含了包的初始化代码，或者设置了 __all__ 变量，后面会有相关介绍。

Users of the package can import individual modules from the package, for example:

包用户可以从包中导入合法的模块，例如

```
import sound.effects.echo
```

This loads the submodule sound.effects.echo. It must be referenced with its full name. :

这样就导入了 Sound.Effects.echo 子模块。它必需通过完整的名称来引用。

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

导入包时有一个可以选择的方式

```
from sound.effects import echo
```

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

这样就加载了 echo 子模块，并且使得它在没有包前缀的情况下也可以 使用，所以它可以如下方式调用

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

还有另一种变体用于直接导入函数或变量

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule echo, but this makes its function echofilter() directly available:

这样就又一次加载了 echo 子模块，但这样就可以直接调用它的 echofilter() 函数

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

需要注意的是使用 from package import item 方式导入包时，这个子项（item）既可以是包中的一个子模块（或一个子包），也可以是包中定义的其它命名，像函数、类或变量。import 语句首先核对是否包中有这个子项，如果没有，它假定这是一个模块，并尝试加载它。如果没有找到它，会引发一个 ImportError 异常。

Contrarily, when using syntax like import item.subitem.subsubitem, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

相反，使用类似 import item.subitem.subsubitem 这样的语法时，这些子项必 须是包，最后的子项可以是包或模块，但不能是前面子项中定义的类、函数或变 量。

6.4.1 Importing * From a Package

Now what happens when the user writes from sound.effects import *? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

那么当用户写下 from sound.Effects import * 时会发生什么事？理想中，总 是希望在文件系统中找出包中所有的子模块，然后导入它们。这可能会花掉委有 长时间，并且出现期待之外的边界效应，导出了希望只能显式导入的包。

The only solution is for the package author to provide an explicit index of the package. The import statement uses the following convention: if a package's __init__.py code defines a list named __all__, it is taken to be the list of module names that should be imported when from package import * is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file sounds/effects/__init__.py could contain the following code:

对于包的作者来说唯一的解决方案就是给提供一个明确的包索引。 import 语句按如下条件进行转换：执行 from package import * 时，如果包中的 __init__.py 代码定义了一个名为 __all__ 的列表，就会按照列表中给出的模块名进行导入。新版本的包发布时作者可以任 意更新这个列表。如果包作者不想 import * 的时候导入他们的包中所有模块， 那么也可能会决定不支持它（import *）。例如， Sounds/Effects/__init__.py 这个文件可能包括如下代码

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that from sound.effects import * would import the three named submodules of the sound package.

这意味着 from Sound.Effects import * 语句会从 sound 包中导入以上三个已命名的子模块。

If __all__ is not defined, the statement from sound.effects import * does not import all submodules from the package sound.effects into the current namespace; it only ensures that the package sound.effects has been imported (possibly running any initialization code in __init__.py) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by __init__.py. It also includes any submodules of the package that were explicitly loaded by previous import statements. Consider this code:

如果没有定义 __all__ ， from Sound.Effects import * 语句不会从 sound.effects 包中导入所有的子模块。无论包中定义多少命名，只能 确定的是导入了 sound.effects 包（可能会运行 __init__.py 中的初 始化代码）以及包中定义的所有命名会随之导入。这样就从 __init__.py 中导入了每一个命名（以及明确导入的子模块）。同样也包括了前述的 import 语句从包中明确导入的子模块，考虑以下代码

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the echo and surround modules are imported in the current namespace because they are defined in the sound.effects package when the from...import statement is executed. (This also works when __all__ is defined.)

在这个例子中， echo 和 surround 模块导入了当前的命名空 间，这是因为执行 from...import 语句时它们已经定义在 sound.effects 包中了（定义了 __all__ 时也会同样工作）。

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practise in production code.

尽管某些模块设计为使用 `import *` 时它只导出符全某种模式的命名，仍然 不建议在生产代码中使用这种写法。

Remember, there is nothing wrong with using `from Package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

记住， `from Package import specific_submodule` 没有错误！事实上，除 非导入的模块需要使用其它包中的同名子模块，否则这是推荐的写法。

6.4.2 Intra-package References 包内引用

The submodules often need to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for a top-level module with the given name.

子模块之间经常需要互相引用。例如，:mod:surround 模块可能会引用 `echo` 模块。事实上，这样的引用如此普遍，以致于 `import` 语句会先搜索包内部，然后才是标准模块搜索路径。因此 `surround` 模 块可以简单的调用 `import echo` 或者 `from echo import echofilter` 。如果没有在当前的包中发现要导入的模块，:keyword:import 语句会依据指 定名寻找一个顶级模块。

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

如果包中使用了子包结构（就像示例中的 `sound` 包），可以按绝对位置 从相邻的包中引入子模块。例如，如果 `sound.filters.vocoder` 包需要 使用 `sound.effects` 包中的 `echo` 模块，它可以 `from Sound.Effects import echo` 。

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

从 Python 2.5 开始，前述的这种内部的显式相对位置导入得到改进，你可以用这样 的形式 `from module import name` 来写显式的相对位置导入。那些显式相 对导入用点号标明关联导入当前和上级包。以 `surround` 模块为例，你可以 这样用

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application should always use absolute imports.

需要注意的是显式或隐式相对位置导入都基于当前模块的命名。因为主模块的名 字总是 `"__main__"` ，Python 应用程序的主模块应该总是用绝对导入。

6.4.3 Packages in Multiple Directories 多重目录中的包

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

包支持一个更为特殊的特性， `__path__` 。 在包的 `__init__.py` 文件代码执行之前，该变量初始化一个目录名列表。该变量可以修改，它作用于 包中的子包和模块的搜索功能。

While this feature is not often needed, it can be used to extend the set of modules found in a package.

这个功能可以用于扩展包中的模块集，不过它不常用。

CHAPTER

SEVEN

INPUT AND OUTPUT 输入和输出

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

有几个方法可以表达程序输出结果；数据可以打印为人类可读的形式，也可以写 入文件供以后使用。本章将讨论几种可选的方法。

7.1 Fancier Output Formatting 玩转输出格式

So far we've encountered two ways of writing values: expression statements and the `print` statement. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

我们有两种大相径庭的输出值方法： 表达式语句 和 `print` 语句。（第三种访求 是使用文件对象的 `write()` 方法，标准文件输出可以参考 `sys.stdout` 。详细内容参见库参考手册。） Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The standard module `string` contains some useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

可能你经常想要对输出格式做一些比简单的打印空格分隔符更为复杂的控制。有 两种方法可以格式化输出。第一种是由你来控制整个字符串，使用字符切割和联 接操作就可以创建出任何你想要的输出形式。标准模块 `string` 包括了一些操 作，将字符串填充入给定列时，这些操作很有用。随后我们会讨论这部分内容。 第二种方法是使用 `str.format()` 方法。

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the `repr()` or `str()` functions.

当然，还有一个问题，如何将值转化为字符串？很幸运，Python 有办法将任意 值转为字符串：将它传入 `repr()` 或 `str()` 函数。

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is not equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

函数 `str()` 用于将值转化为适于人阅读的形式，而 `repr()` 转化为供解释器读 取的形式（如果没有等价的语法，则会发生 `SyntaxError` 异常） 某对象没有适 于人阅读的解释形式的话， `str()` 会返回与 `repr()`

等同的值。很多类型，诸 如数值或链表、字典这样的结构，针对各函数都有着统一的解读方式。字符串和浮点数，有着独特的解读方式。

Some examples:

下面有些例子

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Here are two ways to write a table of squares and cubes:

有两种方式可以写平方和立方表

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Note that in the first example, one space between each column was added by the way print works: it always adds spaces between its arguments.)

(注意第一个例子， print 在每列之间加了一个空格，它总是在参 数间加入空格。)

This example demonstrates the rjust() method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods ljust() and center(). These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in x.ljust(n)[:n].)

以上是一个 rjust() 方法的演示，它把字符串输出到一列，并通过向左 侧填充空格来使其右对齐。类似的方法还有 ljust() 和 center() 。这些函数只是输出新的字符串，并不改变什么。如果输出的字符串太长，它们也不会截断 它，而是原样输出，这会使你的输出格式变得混乱，不过总强过另一种选择(截 断字符串)，因为那样会产生错误的输出值。(如果你确实需要截断它，可以使 用切割操作，例如：x.ljust(n)[:n] 。)

There is another method, zfill(), which pads a numeric string on the left with zeros. It understands about plus and minus signs:

还有另一个方法， zfill() 它用于向数值的字符串表达左侧填充 0。该 函数可以正确理解正负号

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Basic usage of the str.format() method looks like this:

方法 str.format() 的基本用法如下

```
>>> print 'We are the {} who say "{}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the format() method. A number in the brackets refers to the position of the object passed into the format() method. :

大括号和其中的字符会被替换成传入 format() 的参数。大括号中 的数值指明使用传入 format() 方法的对象中的哪一个

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

If keyword arguments are used in the format() method, their values are referred to by using the name of the argument. :

如果在 format() 调用时使用关键字参数，可以通过参数名来引用 值

```
>>> print 'This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

定位和关键字参数可以组合使用

```
>>> print 'The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                    other='Georg')
The story of Bill, Manfred, and Georg.
```

'!s' (apply str()) and '!r' (apply repr()) can be used to convert the value before it is formatted. :

'!s' （应用 str() ） 和 '!r' （应用 repr() ） 可以 在格式化之前转换值。

```
>>> import math
>>> print 'The value of PI is approximately {}.'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

An optional ':' and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example truncates Pi to three places after the decimal.

字段名后允许可选的 ':' 和格式指令。这允许对值的格式化加以更深入的 控制。下例将 Pi 转为三位精度。

```
>>> import math
>>> print 'The value of PI is approximately {0:.3f}.'.format(math.pi)
The value of PI is approximately 3.142.
```

Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making tables pretty. :

在字段后的 ':' 后面加一个整数会限定该字段的最小宽度，这在美化表格时 很有用。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '{0:10} ==> {1:10d}'.format(name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets '[]' to access the keys :

如果你有个实在是很长的格式化字符串，不想分割它。如果你可以用命名来引用 被格式化的变量而不是位置就好了。有个简单的方法，可以传入一个字典，用中 括号访问它的键

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...        'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the `**' notation.

也可以用 ‘**’ 标志将这个字典以关键字参数的方式传入。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the new built-in `vars()` function, which returns a dictionary containing all local variables.

这种方式与新的内置函数 `vars()` 组合使用非常有效。该函数返回包含所有局 部变量的字典。

For a complete overview of string formatting with `str.format()`, see formatstrings.

要进一步了解字符串格式化方法 `str.format()` ，参见 formatstrings 。

7.1.1 Old string formatting 旧式的字符串格式化

The `%` operator can also be used for string formatting. It interprets the left argument much like a `sprintf()`-style format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

操作符 `%` 也可以用于字符串格式化。它以类似 `sprintf()` 的方式 解析左参数，将右参数应用于此，得到格式化操作生成的字符串，例如

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

Since `str.format()` is quite new, a lot of Python code still uses the `%` operator. However, because this old style of formatting will eventually be removed from the language, `str.format()` should generally be used.

因为 `str.format()` 还很新，大量 Python 代码还在使用 `%` 操作符。 然而，因为旧式的格式化方法最终将从语言中去掉，应该尽量使用 :meth:str.format 。

More information can be found in the string-formatting section.

进一步的信息可以参见 :ref:string-formatting 一节。

7.2 Reading and Writing Files 读写文件

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

函数 `open()` 返回文件对象，通常的用法需要两个参数： `open(filename, mode)` 。

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. mode can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The mode argument is optional; `'r'` will be assumed if it's omitted.

第一个参数是一个标识文件名的字符串。第二个参数是由有限的字母组成的字符 串，描述了文件将会被如何使用。可选的 模式 有： 'r' ，此选项使文件只读； 'w' ，此选项使文件只写（对于同名文件，该操作使原有文件被覆盖）； 'a' ， 此选项以追加方式打开文件； 'r+' ，此选项以读写方式打开文件； 模式参数是可选的。如果没有指定，默认为 'r' 模式。

On Windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a 'b' to the mode, so you can use it platform-independently for all binary files.

在Windows 平台上， 'b' 模式以二进制方式打开文件，所以可能会 有类似于 'rb' ， 'wb' ， 'r+b' 等等模式组合。Windows 平台上文本文件与二 进制文件是有区别的，读写文本文件时，行尾会自动添加行结束符。这种后台操 作方式对 ASCII 文本文件没有什么问题，但是操作 JPEG 或 .EXE这样的二进制 文件时就会产生破坏。在操作这些文件时一定要记得以二进制模式打开。在 Unix 上，加一个 'b' 模式也一样是无害的，所以你可以一切二进制文件处 理中平台无关的使用它。

7.2.1 Methods of File Objects 文件对象方法

The rest of the examples in this section will assume that a file object called f has already been created.

本节中的示例都默认文件对象 f 已经创建。

To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string. size is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most size bytes are read and returned. If the end of the file has been reached, f.read() will return an empty string ("").

要读取文件内容，需要调用 f.read(size) ，该方法读取若干数量的数据并以字 符串形式返回其内容，size 是可选的数值，指定字符串长度。如果没有指定 size或者指定为负数，就会读取并返回整个文件。当文件大小为当前机器内存两 倍时，就会产生问题。反之，会尽可能按比较大的 size 读取和返回数据。 如果到了文件末尾，f.read()会返回一个空字符串（''``）。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

f.readline() 从文件中读取单独一行，字符串结尾会自动加上一个换行符 （ \n ），只有当文件最后一行没有以换行符结尾时，这一操作才会被忽略。 这样返回值就不会有混淆，如果如果 f.readline() 返回一个空字符串，那就表 示到达了文件末尾，如果是一个空行，就会描述为 '\n ，一个只包含换行符的字符串。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
```

```
>>> f.readline()
''
```

`f.readlines()` returns a list containing all the lines of data in the file. If given an optional parameter sizehint, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that. This is often used to allow efficient reading of a large file by lines, but without having to load the entire file in memory. Only complete lines will be returned.

f.readlines()返回一个列表，其中包含了文件中所有的数据行。如果给定了 sizehint 参数，就会读入多于一行的比特数，从中返回多行文本。这个功能 通常用于高效读取大型行文件，避免了将整个文件读入内存。这种操作只返回完 整的行。

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

An alternative approach to reading lines is to loop over the file object. This is memory efficient, fast, and leads to simpler code

有个按行读取的好办法是在文件对象上循环。这样容易记忆，高速而且代码更简单

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

The alternative approach is simpler but does not provide as fine-grained control. Since the two approaches manage line buffering differently, they should not be mixed.

这个办法很简单，但不能完整的控制操作。因为两个方法用不同的方式管理行缓 冲区，它们不能混用。

`f.write(string)` writes the contents of string to the file, returning `None`.

`f.write(string)` 将 string 的内容写入文件，返回 `None` 。

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

如果需要写入字符串以外的数据，就要先把这些数据转换为字符串。

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding offset to a reference point; the reference point is selected by the from_what argument. A from_what value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. from_what can be omitted and defaults to 0, using the beginning of the file as the reference point.

`f.tell()` 返回一个整数，代表文件对象在文件中的指针位置，该数值计量了自文 件开头到指针处的比特数。需要改变文件对象指针话话，使用 `f.seek(offset,from_what)` 。指针在该操作中从指定的引用位置移动 offset 比特，引用位置由 from_what 参数指定。 from_what 值为 0 表示自文件 起始处开始，1 表示自当前文件指针位置开始，2 表示自文件末尾开始。 from_what 可以 忽略，其默认值为零，此时从文件头开始。

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

文件使用完后，调用 `f.close()` 可以关闭文件，释放打开文件后占用的系统资源。 调用 `f.close()` 之后，再调用文件对象会自动引发错误。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks

用关键字 `with` 处理文件对象是个好习惯。它的先进之处在于文件 用完后会自动关闭，就算发生异常也没关系。它是 `try-finally` 块的简写。

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

文件对象还有一些不太常用的附加方法，比如 `isatty()` 和 `truncate()` 在库参 考手册中有文件对象的完整指南。

7.2.2 The `pickle` Module `pickle` 模块

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. However, when you want to save more complex data types like lists, dictionaries, or class instances, things get a lot more complicated.

我们可以很容易的读写文件中的字符串。数值就要多费点儿周折，因为 `read()` 方法只会返回字符串，应该将其传入 `int()` 这样的方法中，就可以将 `'123'` 这样的字符转为对应的数值 123 。不过，当你需要保存更为复杂的 数据类型，例如列表、字典，类的实例，事情就会变得更复杂了。

Rather than have users be constantly writing and debugging code to save complicated data types, Python provides a standard module called `pickle`. This is an amazing module that can take almost any Python object (even some forms of Python code!), and convert it to a string representation; this process is called pickling. Reconstructing the object from the string representation is called unpickling. Between pickling and unpickling, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

好在用户不必要非得自己编写和调试保存复杂数据类型的代码。 Python提供了 一个名为 pickle 的标准模块。这是一个令人赞叹的模块，几乎可以把任何 Python 对象 （甚至是一些 Python 代码段！）表达为为字符串，这一过程称之 为封装 （ pickling ）。从字符串表达出重新构造对象称之为拆封 （ unpickling ）。封装状态中的对象可以存储在文件或对象中，也可以通过网 络在远程的机器之间传输。

If you have an object x, and a file object f that's been opened for writing, the simplest way to pickle the object takes only one line of code

如果你有一个对象 x ，一个以写模式打开的文件对象 f ，封装对象的最简单的 方法只需要一行代码

```
pickle.dump(x, f)
```

To unpickle the object again, if f is a file object which has been opened for reading

如果 f 是一个以读模式打开的文件对象，就可以重装拆封这个对象

```
x = pickle.load(f)
```

(There are other variants of this, used when pickling many objects or when you don't want to write the pickled data to a file; consult the complete documentation for pickle in the Python Library Reference.)

（如果不想把封装的数据写入文件，这里还有一些其它的变化可用。完整的 pickle 文档请见Python 库参考手册）。

pickle is the standard way to make Python objects which can be stored and reused by other programs or by a future invocation of the same program; the technical term for this is a persistent object. Because pickle is so widely used, many authors who write Python extensions take care to ensure that new data types such as matrices can be properly pickled and unpickled.

pickle 是存储 Python 对象以供其它程序或其本身以后调用的标准方法。提供 这一组技术的是一个 持久化 对象（ persistent object ）。因为 pickle 的用 途很广泛，很多 Python 扩展的作者都非常注意类似矩阵这样的新数据类型是否适合封装和拆封。

CHAPTER

EIGHT

ERRORS AND EXCEPTIONS 错误和异常

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.

至今为止还没有进一步的谈论过错误信息，不过在你已经试验过的那些例子中， 可能已经遇到过一些。Python 中（至少）有两种错误：语法错误和异常 （ syntax errors and exceptions ）。

8.1 Syntax Errors 语法错误

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

语法错误，也称作解释错误，可能是学习 Python 的过程中最容易犯的

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little `arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the keyword `print`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

解析器会重复出错的行，并在行中最早发现的错误位置上显示一个小"箭头"。错误 （至少是被检测到的）就发生在箭头 指向 的位置。示例中的错误表现在关键字 `print` 上，因为在它之前少了一个冒号（ `':'` ）。同时也会显示文件名和行号， 这样你就可以知道错误来自哪个脚本，什么位置。

8.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

即使是在语法上完全正确的语句，尝试执行它的时候，也有可能会发生错误。在 程序运行中检测出的错误称之为异常，它通常不会导致致命的问题，你很快就会 学到如何在 Python 程序中控制它们。大多数异常不会由程序处理，而是显示一 个错误信息

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

错误信息的最后一行指出发生了什么错误。异常也有不同的类型，异常类型做为 错误信息的一部分显示出来：示例中的异常分别为 零除错误 （ ZeroDivisionError ） ，命名错误（ NameError` :exc:`TypeError ）。打印错误信息时，异常的类型作为异常的内置名 显示。对于所有的内置异常都是如此，不过用户自定义异常就不一定了（尽管这 是一个很有用的约定）。标准异常名是内置的标识（没有保留关键字）。

The rest of the line provides detail based on the type of exception and what caused it.

这一行后一部分是关于该异常类型的详细说明，这意味着它的内容依赖于异常类型。

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

错误信息的前半部分以堆栈的形式列出异常发生的位置。通常在堆栈中列出了源代码行，然而，来自标准输入的源码不会显示出来。

bltin-exceptions lists the built-in exceptions and their meanings.

bltin-exceptions 列出了内置异常和它们的含义。

8.3 Handling Exceptions 控制异常

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception. :

可以编写程序来控制已知的异常。参见下例，此示例要求用户输入信息，一直到 得到一个有效的整数为止，而且允许用户中断程序（使用 Control-C 或 其它什么操作系统支持的操作）；需要注意的是用户生成的中断会抛出 KeyboardInterrupt 异常。

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
```

```
...         print "Oops!  That was no valid number.  Try again..."
...
```

The try statement works as follows.

try 语句按如下方式工作。

- First, the try clause (the statement(s) between the try and except keywords) is executed.

  首先，执行 try 子句 （在 try 和 except 关键字之间的部分）。

- If no exception occurs, the except clause is skipped and execution of the try statement is finished.

  如果没有异常发生， except 子句 在 try 语句执行完毕后就被忽略了。

- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

  如果在 try 子句执行过程中发生了异常，那么该子句其余的部分就会被忽略。 如果异常匹配于 except 关键字后面指定的异常类型，就执行对应的except子 句。然后继续执行 try 语句之后的代码。

- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

  如果发生了一个异常，在 except 子句中没有与之匹配的分支，它就会传递到 上一级 try 语句中。 如果最终仍找不到对应的处理语句，它就成 为一个 未处理异常 ，终止程序运行，显示提示信息。

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

一个 try 语句可能包含多个 except 子句，分别指定处理不同的异 常。至多只会有一个分支被执行。异常处理程序只会处理对应的 try 子句中发 生的异常，在同一个 try 语句中，其他子句中发生的异常则不作处 理。一个except子句可以在括号中列出多个异常的名字，例如

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

最后一个 except 子句可以省略异常名，把它当做一个通配项使用。一定要慎用 这种方法，因为它很可能会屏蔽掉真正的程序错误，使人无法发现！它也可以用 于打印一行错误信息，然后重新抛出异常（可以使调用者更好的处理异常）

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
```

```
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

The try ... except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

try ... except 语句可以带有一个 else 子句 ， 该子句只能出现在所有 except 子句之后。当 try 语句没有抛出异常时，需要执行一些代码，可以使用 这个子句。例如

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement.

使用 else 子句比在 try 子句中附加代码要好，因为 这样可以避免 try ... except 意外的截获本来不属于 它们保护的那些代码抛出的异常。

When an exception occurs, it may have an associated value, also known as the exception's argument. The presence and type of the argument depend on the exception type.

发生异常时，可能会有一个附属值，作为异常的 参数 存在。这个参数是否存在、 是什么类型，依赖于异常的类型。

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines __str__() so the arguments can be printed directly without having to reference .args.

在异常名（列表）之后，也可以为 except 子句指定一个变量。这个变量绑定于 一个异常实例，它存储在 instance.args 的参数中。为了方便起见，异常实例 定义了 __str__() ，这样就可以直接访问过打印参数而不必引用 .args 。

One may also instantiate an exception first before raising it and add any attributes to it as desired. :

这种做法不受鼓励。相反，更好的做法是给异常传递一个参数（如果要传递多个 参数，可以传递一个元组），把它绑定到 message 属性。一旦异常发生，它会 在抛出前绑定所有指定的属性。

```
>>> try:
...    raise Exception('spam', 'eggs')
... except Exception as inst:
...    print type(inst)     # the exception instance
...    print inst.args      # arguments stored in .args
...    print inst           # __str__ allows args to printed directly
...    x, y = inst          # __getitem__ allows args to be unpacked directly
...    print 'x =', x
...    print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
```

```
x = spam
y = eggs
```

If an exception has an argument, it is printed as the last part (`detail') of the message for unhandled exceptions.

对于未处理的异常，如果它有一个参数，那做就会作为错误信息的最后一部分 （“明细”）打印出来。

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

异常处理句柄不止可以处理直接发生在 try 子句中的异常，即使是其中（甚至 是间接）调用的函数，发生了异常，也一样可以处理。例如

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

8.4 Raising Exceptions 抛出异常

The `raise` statement allows the programmer to force a specified exception to occur. For example:

程序员可以用 `raise` 语句强制指定的异常发生。例如

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`).

要抛出的异常由 `raise` 的唯一参数标识。它必需是一个异常实例或 异常类（继承自 `Exception` 的类）。

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

如果你需要明确一个异常是否抛出，但不想处理它， `raise` 语句可以让你很简单的重新抛出该异常。

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5 User-defined Exceptions 用户自定义异常

Programs may name their own exceptions by creating a new exception class (see Classes 类 for more about Python classes). Exceptions should typically be derived from the Exception class, either directly or indirectly. For example:

在程序中可以通过创建新的异常类型来命名自己的异常（Python 类的内容请参 见 Classes 类 ）。异常类通常应该直接或间接的从 Exception 类派生，例如

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

In this example, the default __init__() of Exception has been overridden. The new behavior simply creates the value attribute. This replaces the default behavior of creating the args attribute.

在这个例子中，:class:Exception 默认的 __init__() 被覆盖。新的方式简单的创建 value 属性。这就替换了原来创建 args 属性的方式。

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

异常类中可以定义任何其它类中可以定义的东西，但是通常为了保持简单，只在 其中加入几个属性信息，以供异常处理句柄提取。如果一个新创建的模块中需要 抛出几种不同的错误时，一个通常的作法是为该模块定义一个异常基类，然后针 对不同的错误类型派生出对应的异常子类。

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg
```

```
class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """

    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg
```

Most exceptions are defined with names that end in ``Error,'' similar to the naming of the standard exceptions.

与标准异常相似，大多数异常的命名都以“Error”结尾。

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter Classes 类.

很多标准模块中都定义了自己的异常，用以报告在他们所定义的函数中可能发生 的错误。关于类的进一步信息请参见 Classes 类 一章。

8.6 Defining Clean-up Actions 定义清理行为

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

`try` 语句还有另一个可选的子句，目的在于定义在任何情况下都一定要执行的功 能。例如

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
```

A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in a `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed ``on the way out'' when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

不管有没有发生异常， `finally` 子句 在程序离开 `try` 后都一定 会被执行。当 `try` 语句中发生了未被 `except` 捕获的 异常（或者它发生在 `except` 或 `else` 子句中），在 `finally` 子句执行完后它会被重新抛出。 `try` 语句经 由 `break` ，:keyword:continue 或 `return` 语句退 出也一样会执行 `finally` 子句。以下是一个更复杂些的例子（在同 一个 `try` 语句中的 `except` 和 `finally` 子句的工作方式与 Python 2.5 一样）

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

如你所见， `finally` 子句在任何情况下都会执 行。 `TypeError` 在两个字符串相除的时候抛出，未被 `except` 子句捕获，因此在 `finally` 子句执行完毕后重新抛出。

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

在真实场景的应用程序中， `finally` 子句用于释放外部资源（文件 或网络连接之类的），无论它们的使用过程中是否出错。

8.7 Predefined Clean-up Actions 预定义清理行为

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen. :

有些对象定义了标准的清理行为，无论对象操作是否成功，不再需要该对象的时 候就会起作用。以下示例尝试打开文件并把内容打印到屏幕上。

```
for line in open("myfile.txt"):
    print line
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly. :

这段代码的问题在于在代码执行完后没有立即关闭打开的文件。这在简单的脚本 里没什么，但是大型应用程序就会出问题。 `with` 语句使得文件之类的对象可以 确保总能及时准确地进行清理。

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

After the statement is executed, the file f is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

语句执行后，文件 f 总会被关闭，即使是在处理文件中的数据时出错也一样。 其它对象是否提供了预定义的清理行为要查看它们的文档。

CHAPTER

NINE

CLASSES 类

Python's class mechanism adds classes to the language with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. As is true for modules, classes in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to ``break into the definition.'' The most important features of classes are retained with full power, however: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of data.

Python 在尽可能不增加新的语法和语义的情况下加入了类机制。这种机制是 C++ 和 Modula-3 的混合。像模块那样， Python 中的类没有在用户和定义之间建立一个绝对的 屏障，而是依赖于用户自觉的不去“破坏定义”。然而，类机制最重要的功能都 完整的保留下来：类继承机制允许多继承，派生类可以覆盖（override）基类中 的任何方法，方法中可以调用基类中的同名方法。对象可以包含任意数量的数据。

In C++ terminology, all class members (including the data members) are public, and all member functions are virtual. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

用 C++ 术语来讲，所有的类成员（包括数据成员）都是公有（ public ）的， 所有的成员函数都是 虚 （ virtual ）的。用 Modula-3的术语来讲，在成员方法中没有简便的方式引 用对象的成员：方法函数在定义时需要以引用的对象做为第一个参数，调用时则 会隐式引用对象。像在 Smalltalk 中一样，类也是对象。这就提供了导入和重 命名语义。不像 C++ 和 Modula-3 中 那样，大多数带有特殊语法的内置操作符（算法运算符、下标等）都可以针对类 的需要重新定义。

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

（在讨论类时，没有足够的得到共识的术语，我会偶尔从 Smalltalk 和 C++ 借用一些。我比较喜欢用 Modula-3 的用语，因为比起C++， Python 的面 向对象语法更像它，但是我想很少有读者听过这个。）

9.1 A Word About Names and Objects 关于命名和对象的内容

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change --- this eliminates the need for two different argument passing mechanisms as in Pascal.

对象是被特化的，多个名字（在多个作用域中）可以绑定同一个对象。这相当于 其它语言中的别名。通常对 Python 的第一印象中会忽略这一点，使用那些不可 变的基本类型（数值、字符串、元组）时也可以很放心的忽视它。然而，在 Python 代码调用字典、列表之类可变对象，以及大多数涉及程序外部实体（文件、窗体等等）的类型时，这一语义就会有影响。这通用有助于优化程序，因为 别名的行为在某些方面类似于指针。例如，很容易传递一个对象，因为在行为上 只是传递了一个指针。如果函数修改了一个通过参数传递的对象，调用者可以接 收到变化——在 Pascal 中这需要两个不同的参数传递机制。

9.2 Python Scopes and Namespaces Python 作用域和命名空间

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

在介绍类之前，我首先介绍一些有关 Python 作用域的规则。类的定义非常巧妙 的运用了命名空间，要完全理解接下来的知识，需要先理解作用域和命名空间的 工作原理。另外，这一切的知识对于任何高级 Python 程序员都非常有用。

Let's begin with some definitions.

我们从一些定义开始。

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (functions such as abs(), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion --- users of the modules must prefix it with the module name.

命名空间 是从命名到对象的映射。当前命名空间主要是通过 Python 字典实现 的，不过通常不关心具体的实现方式（除非出于性能考虑），以后也有可能会改 变其实现方式。以下有一些命名空间的例子：内置命名（像 abs() 这样的函数，以 及内置异常名）集，模块中的全局命名，函数调用中的局部命名。某种意义上讲 对象的属性集也是一个命名空间。关于命名空间需要了解的一件很重要的事就是 不同命名空间中的命名没有任何联系，例如两个不同的模块可能都会定义一个名 为 maximize 的函数而不会发生混淆——用户必须以模块名为前缀来引用它们。

By the way, I use the word attribute for any name following a dot --- for example, in the expression z.real, real is an attribute of the object z. Strictly speaking, references to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

[1] Except for one thing. Module objects have a secret read-only attribute called __dict__ which returns the dictionary used to implement the module's namespace; the name __dict__ is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

顺便提一句，我称 Python 中任何一个“.”之后的命名为 属性 ——例如，表达 式 z.real 中的 real 是对象 z 的一个属性。严格来讲，从模块中引用命名是 引用属性：表达式 modname.funcname 中， modname 是一个模块对象，funcname 是它的一个属性。因此，模块的属性和模块中的全局命名有直接的映射关系：它 们共享同一命名空间！[2]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write modname.the_answer = 42. Writable attributes may also be deleted with the del statement. For example, del modname.the_answer will remove the attribute the_answer from the object named by modname.

属性可以是只读过或写的。后一种情况下，可以对属性赋值。你可以这样 作： modname.the_answer = 42 。可写的属性也可以用 del 语句删除。例 如： del modname.the_answer 会从 modname 对象中删除 the_answer 属性。

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called __main__, so they have their own global namespace. (The built-in names actually also live in a module; this is called __builtin__.)

不同的命名空间在不同的时刻创建，有不同的生存期。包含内置命名的命名空间 在 Python 解释器启动时创建，会一直保留，不被删除。模块的全局命名空间在 模块定义被读入时创建，通常，模块命名空间也会一直保存到解释器退出。由解 释器在最高层调用执行的语句，不管它是从脚本文件中读入还是来自交互式输 入，都是 __main__ 模块的一部分，所以它们也拥有自己的命名空间。（内置命 名也同样被包含在一个模块中，它被称作 __builtin__ 。）

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

当函数被调用时创建一个局部命名空间，函数反正返回过抛出一个未在函数内处 理的异常时删除。（实际上，说是遗忘更为贴切）。当然，每一个递归调用拥有 自己的命名空间。

A scope is a textual region of a Python program where a namespace is directly accessible. ``Directly accessible'' here means that an unqualified reference to a name attempts to find the name in the namespace.

作用域 是Python程序中一个命名空间可以直接访问的文法区域。“直接访问” 在这里的意思是查找命名时无需引用命名前缀。

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

尽管作用域是静态定义，在使用时他们都是动态的。每次执行时，至少有三个命 名空间可以直接访问的作用域嵌套在一起：包含局部命名的使用域在最里面，首 先被搜索；其次搜索的是中层的作用域，这里包含了同级的函数；最后搜索最外 面的作用域，它包含内置命名。

- the innermost scope, which is searched first, contains the local names

  首先搜索最内层的作用域，它包含局部命名

- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names

  任意函数包含的作用域，是内层嵌套作用域搜索起点，包含非局部，但是也非 全局的命名

[2] 有一个例外。模块对象有一个隐秘的只读对象，名为 __dict__ ，它 返回用于实现模块命名空间的字典，命名 __dict__ 是一个属性而非 全局命名。显然，使用它违反了命名空间实现的抽象原则，应该被严格限制于 调试中。

- the next-to-last scope contains the current module's global names

  接下来的作用域包含当前模块的全局命名

- the outermost scope (searched last) is the namespace containing built-in names

  最外层的作用域（最后搜索）是包含内置命名的命名空间。

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

如果一个命名声明为全局的，那么所有的赋值和引用都直接针对包含模全局命名 的中级作用域。另外，从外部访问到的所有内层作用域的变量都是只读的。（试 图写这样的变量只会在内部作用域创建一个 新 局部变量，外部标示命名的那 个变量不会改变）。

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

通常，局部作用域引用当前函数的命名。在函数之外，局部作用域与 全局使用域引用同一命名空间：模块命名空间。类定义也是局部作用域中的另一 个命名空间。

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time --- however, the language definition is evolving towards static name resolution, at ``compile'' time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

重要的是作用域决定于源程序的意义：一个定义于某模块中的函数的全局作用域 是该模块的命名空间，而不是该函数的别名被定义或调用的位置，了解这一点非 常重要。另一方面，命名的实际搜索过程是动态的，在运行时确定的——然 而，Python 语言也在不断发展，以后有可能会成为静态的“编译”时确定，所以不要依赖动态解析！（事实上，局部变量已经是静态确定了。）

A special quirk of Python is that -- if no `global` statement is in effect -- assignments to names always go into the innermost scope. Assignments do not copy data --- they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

Python 的一个特别之处在于——如果没有使用 `global` 语法——其赋值 操作总是在最里层的作用域。赋值不会复制数据——只是将命名绑定到对象。删除 也是如此： `del x` 只是从局部作用域的命名空间中删除命名 `x` 。事实 上，所有引入新命名的操作都作用于局部作用域。特别是 `import` 语句和函数定将模块名或函数绑定于局部作用域。（可以使用 `global` 语句将变量引入到全局作用域。）

9.3 A First Look at Classes 初识类

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

类引入了一点新的语法，三种新的对象类型，以及一些新的语义。

9.3.1 Class Definition Syntax 类定义语法

The simplest form of class definition looks like this:

最简单的类定义形式如下

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

类的定义就像函数定义（ def 语句），要先执行才能生效。（你当然可以把它 放进 if 语句的某一分支，或者一个函数的内部。）

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful --- we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods --- again, this is explained later.

习惯上，类定义语句的内容通常是函数定义，不过其它语句也可以，有时会很有 用——后面我们再回过头来讨论。类中的函数定义通常包括了一个特殊形式的参数 列表，用于方法调用约定——同样我们在后面讨论这些。

When a class definition is entered, a new namespace is created, and used as the local scope --- thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域——因此，所有 的赋值成为这个新命名空间的局部变量。特别是函数定义在此绑定了新的命名。

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

类定义完成时（正常退出），就创建了一个 类对象 。基本上它是对类定义创建的 命名空间进行了一个包装；我们在下一节进一步学习类对象的知识。原始的局部 作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头 部的类名（例子中是 ClassName ）。

9.3.2 Class Objects 类对象

Class objects support two kinds of operations: attribute references and instantiation.

类对象支持两种操作：属性引用和实例化。

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

属性引用使用和 Python 中所有的属性引用一样的标准语法：obj.name。类对象 创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. __doc__ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

那么 MyClass.i 和 MyClass.f 是有效的属性引用，分别返回一个整数和一个方 法对象。也可以对类属性赋值，你可以通过给 MyClass.i 赋值来修改它。 __doc__` 也是一个有效的属性，返回类的文档字符串： "A simple example class" 。

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

类的 实例化 使用函数符号。只要将类对象看作是一个返回新的类实例的无参 数函数即可。例如（假设沿用前面的类）

```
x = MyClass()
```

creates a new instance of the class and assigns this object to the local variable x.

以上创建了一个新的类 实例 并将该对象赋给局部变量 x 。

The instantiation operation (``calling'' a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

这个实例化操作（“调用”一个类对象）来创建一个空的对象。很多类都倾向于 将对象创建为有初始状态的。因此类可能会定义一个名为 __init__() 的特殊方 法，像下面这样

```
def __init__(self):
    self.data = []
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

类定义了 __init__() 方法的话，类的实例化操作会自动为新创建的类实例调用 __init__`() 方法。所以在下例中，可以这样创建一个新的实例

```
x = MyClass()
```

Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example, :

当然，出于弹性的需要， __init__() 方法可以有参数。事实上，参数通过 __init__`() 传递到类的实例化操作上。例如

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Instance Objects 实例对象

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

现在我们可以用实例对象作什么？实例对象唯一可用的操作就是属性引用。有两 种有效的属性名。

data attributes correspond to ``instance variables'' in Smalltalk, and to ``data members'' in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

数据属性 相当于 Smalltalk 中的“实例变量”或 C++中的“数据成员”。和局 部变量一样，数据属性不需要声明，第一次使用时它们就会生成。例如，如果 x 是前面创建的 MyClass 实例，下面这段代码会打印出 16 而在堆 栈中留下多余的东西

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

The other kind of instance attribute reference is a method. A method is a function that ``belongs to'' an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

另一种为实例对象所接受的引用属性是 方法 。方法是“属于”一个对象的函数。 （在 Python 中，方法不止是类实例所独有：其它类型的对象也可有方法。例 如，链表对象有 append，insert，remove，sort 等等方法。然而，在后面的介 绍中，除非特别说明，我们提到的方法特指类方法） Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.f is not the same thing as MyClass.f --- it is a method object, not a function object.

实例对象的有效名称依赖于它的类。按照定义，类中所有（用户定义）的函数对 象对应它的实例中的方法。所以在我们的例子中，``x.f`` 是一个有效的方法引用， 因为 MyClass.f 是一个函数。但 x.i 不是，因为 MyClass.i 不是函数。不 过 x.f 和 MyClass.f 不同——它是一个 方法对象 ，不是一个函数对象。

9.3.4 Method Objects 方法对象

Usually, a method is called right after it is bound:

通常，方法通过右绑定调用

```
x.f()
```

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

在 MyClass 示例中，这会返回字符串 'hello world' 。然而，也不是一定要直 接调用方法。 x.f 是一个方法对象，它可以存储起来以后调用。例如

```
xf = x.f
while True:
    print xf()
```

will continue to print `hello world` until the end of time.

会不断的打印 ``hello world'' 。

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any --- even if the argument isn't actually used...

调用方法时发生了什么？你可能注意到调用 `x.f()` 时没有引用前面标出的变量，尽 管在 `f()` 的函数定义中指明了一个参数。这个参数怎么了？事实上如果函数调用 中缺少参数，Python 会抛出异常——甚至这个参数实际上没什么用……

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

实际上，你可能已经猜到了答案：方法的特别之处在于实例对象作为函数的第一 个参数传给了函数。在我们的例子中，调用 `x.f()` 相当于 `MyClass.f(x)` 。通 常，以 n 个参数的列表去调用一个方法就相当于将方法的对象插入到参数列表 的最前面后，以这个列表去调用相应的函数。

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

如果你还是不理解方法的工作原理，了解一下它的实现也许有帮助。引用非数据 属性的实例属性时，会搜索它的类。如果这个命名确认为一个有效的函数对象类 属性，就会将实例对象和函数对象封装进一个抽象对象：这就是方法对象。以一 个参数列表调用方法对象时，它被重新拆封，用实例对象和原始的参数列表构造 一个新的参数列表，然后函数对象调用这个新的参数列表。

9.4 Random Remarks 一些说明

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

同名的数据属性会覆盖方法属性，为了避免可能的命名冲突——这在大型程序中 可能会导致难以发现的 bug ——最好以某种命名约定来避免冲突。可选的约定 包括方法的首字母大写，数据属性名前缀小写（可能只是一个下划线），或者方 法使用动词而数据属性使用名词。

Data attributes may be referenced by methods as well as by ordinary users (``clients'') of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding --- it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

数据属性可以由方法引用，也可以由普通用户（客户）调用。换句话说，类不能 实现纯抽象数据类型。事实上 Python 中没有什么办法可以强制隐藏数据——一切 都基本约定的惯例。（另一方法讲，Python 的实现是用 C 写成的，如果有必 要，可以用 C 来编写 Python 扩展，完全隐藏实现的细节，控制对象的访问。）

Clients should use data attributes with care --- clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided --- again, a naming convention can save a lot of headaches here.

客户应该小心使用数据属性——客户可能会因为随意修改数据属性而破坏了本来 由方法维护的数据一致性。需要注意的是，客户只要注意避免命名冲突，就可以 随意向实例中添加数据属性而不会影响方法的有效性——再次强调，命名约定可 以省去很多麻烦。

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

从方法内部引用数据属性（以及其它方法！）没有什么快捷的方式。我认为这事 实上增加了方法的可读性：即使粗略的浏览一个方法，也不会有混淆局部变量和 实例变量的机会。

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

通常方法的第一个参数命名为 `self` 。这仅仅是一个约定：对 Python 而言， `self` 绝对没有任何特殊含义。（然而要注意的是，如果不遵守这个约定，别的 Python 程序员阅读你的代码时会有不便，而且有些类浏览程序也是遵循此约定 开发的。）

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

类属性中的任何函数对象在类实例中都定义为方法。不是必须要将函数定义代码 写进类定义中，也可以将一个函数对象赋给类中的一个变量。例如

```python
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` --- `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

现在 `f`, `g` 和 `h` 都是类 `C` 的属性，引用的都是函数对 象，因此它们都是 `C` 实例的方法—— `h` 严格等于 `g` 。要注意的是这 种习惯通常只会迷惑程序的读者。

Methods may call other methods by using method attributes of the `self` argument:

通过 `self` 参数的方法属性，方法可以调用其它的方法

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing the class definition. (The class itself is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

方法可以像引用普通的函数那样引用全局命名。与方法关联的全局作用域是包含 类定义的模块。（类本身永远不会做为全局作用域使用。）尽管很少有好的理由 在方法中使用全局数据，全局作用域确有很多合法的用途：其一是方法可以调用 导入全局作用域的函数和方法，也可以调用定义在其中的类和函数。通常，包含 此方法的类也会定义在这个全局作用域，在下一节我们会了解为何一个方法要引 用自己的类。

Each value is an object, and therefore has a class (also called its type). It is stored as object.__class__.

每个值都是一个对象，因此每个值都有一个 类(class) （也称为它的 类型(type) ），它存储为 object.__class__ 。

9.5 Inheritance 继承

Of course, a language feature would not be worthy of the name ``class'' without supporting inheritance. The syntax for a derived class definition looks like this:

当然，如果一种语言不支持继承就，“类”就没有什么意义。派生类的定义如下 所示

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

命名 BaseClassName （示例中的基类名）必须与派生类定义在一个作用域内。除 了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

派生类定义的执行过程和基类是一样的。构造派生类对象时，就记住了基类。这 在解析属性引用的时候尤其有用：如果在类中找不到请求调用的属性，就搜索基 类。如果基类是由别的类派生而来，这个规则会递归的应用上去。

There's nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

派生类的实例化没有什么特殊之处： DerivedClassName() （示列中的派生类） 创建一个新的类实例。方法引用按如下规则解析：搜索对应的类属性，必要时沿 基类链逐级搜索，如果找到了函数对象这个方法引用就是合法的。

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

派生类可能会覆盖其基类的方法。因为方法调用同一个对象中的其它方法时没有 特权，基类的方法调用同一个基类的方法时，可能实际上最终调用了派生类中的 覆盖方法。（对于 C++ 程序员来说，Python中的所有方法本质上都是 方法。）

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments). This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as BaseClassName in the global scope.)

派生类中的覆盖方法可能是想要扩充而不是简单的替代基类中的重名方法。有一 个简单的方法可以直接调用基类方法，只要调用： BaseClassName.methodname(self, arguments) 。有时这对于客户也很有用。（要注意只有 BaseClassName 在同一全局作用域定义或导入时才能这样用。）

Python has two built-in functions that work with inheritance:

Python 有两个用于继承的函数：

- Use isinstance() to check an instance's type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.

  函数 isinstance() 用于检查实例类型： isinstance(obj, int) 只有在 obj.__class__ 是 int 或其它从 int 继承 的类型

- Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(unicode, str) is False since unicode is not a subclass of str (they only share a common ancestor, basestring).

  函数 issubclass() 用于检查类继承： issubclass(bool, int) 为 True ，因为 bool 是 int 的子类。但是， issubclass(unicode, str) 是 False ，因为 unicode 不 是 str 的子类（它们只是共享一个通用祖先类 basestring ）。

9.5.1 Multiple Inheritance 多继承

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

Python同样有限的支持多继承形式。多继承的类定义形如下例

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

对于旧式的类，唯一的规则顺序是深度优先，从左到右。因此，如果在 DerivedClassName （示例中的派生类）中没有找到某个属性，就会搜 索 Base1 ，然后（递归的）搜索其基类，如果最终没有找到，就搜索 Base2 ，以此类推。

(To some people breadth first --- searching Base2 and Base3 before the base classes of Base1 --- looks more natural. However, this would require you to know whether a particular attribute of Base1 is actually defined in Base1 or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of Base2. The depth-first rule makes no differences between direct and inherited attributes of Base1.)

（有些人认为广度优先——在搜索 Base1 的基类之前搜 索:class:Base2 和 Base3 ——看起来更为自然。然而，如果 Base1 和 Base2 之间发生了命名冲突，你需要了解这个属性 是定义于 :Base1 还是 Base1 的基类中。而深度优先不区分 属性继承自基类还是直接定义。）

For new-style classes, the method resolution order changes dynamically to support cooperative calls to super(). This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

对于 :term:new-style class 类， super() 可以动态的改变解析顺 序。这个方式可见于其它的一些多继承语言，类似 call-next-method，比单继 承语言中的 super 更强大 ：

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where one at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from object, so any case of multiple inheritance provides more than one path to reach object. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see http://www.python.org/download/releases/2.3/mro/ .

在 new-style 类中，必须有动态调整顺序，因为所有的多继承会有一到多个菱 形关系（指有至少一个祖先类可以从子类经由多个继承路径到达）。例如，所有 的 new-style 类继承自 object ，所以任意的多继承总是会有多于一 条继承路径到达 object 。为了防止重复访问基类，通过动态的线性化 算法，每个类都按从左到右的顺序特别指定了顺序，每个祖先类只调用一次，这 是单调的（意味着一个类被继承时不会影响它祖先的次序）。总算可以通过这种 方式使得设计一个可靠并且可扩展的多继承类成为可能。进一步的内容请参见 http://www.python.org/download/releases/2.3/mro/ 。

9.6 Private Variables 私有变量

``Private'' instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

只能从对像内部访问的“私有”实例变量，在 Python 中不存在。然而，也有 一个变通的访问用于大多数 Python 代码：以一个下划线开头的命名（例如 _spam ）会被处理为 API 的非公开部分（无论它是一个函

数、方法或数据 成员）。它会被视为一个实现细节，无需公开。

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

因为有一个正当的类私有成员用途（即避免子类里定义的命名与之冲突），Python 提供了对这种结构的有限支持，称为 name mangling（命名编码） 。任何形如 `__spam` 的标识（前面至少两个下划线，后面至多一个），被替代为 `_classname__spam` ，去掉前导下划线的 `classname` 即当前的类名。此 语法不关注标识的位置，只要求在类定义内。

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

需要注意的是编码规则设计为尽可能的避免冲突，被认作为私有的变量仍然有可 能被访问或修改。在特定的场合它也是有用的，比如调试的时候。

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

要注意的是代码传入 `exec` ， `eval()` 或 `execfile()` 时不考虑所调 用的类的类名，视其为当前类，这类似于 `global` 语句的效应， 已经按字节编译的部分也有同样的限制。这也同样作用于 `getattr()` ，`setattr()` 和 `delattr` ，像直接引用 `__dict__` 一样。

9.7 Odds and Ends 补充

Sometimes it is useful to have a data type similar to the Pascal ``record'' or C ``struct'', bundling together a few named data items. An empty class definition will do nicely:

有时类似于Pascal中“记录（record）”或C中“结构（struct）”的数据类型 很有用，它将一组已命名的数据项绑定在一起。一个空的类定义可以很好的实现 这它

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

某一段 Python 代码需要一个特殊的抽象数据结构的话，通常可以传入一个类，事实上这模仿了该类的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有 `read()` 和 `readline()` 方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

实例方法对象也有属性： `m.im_self` 是一个实例方法所属的对象，而 `m.im_func` 是这个方法对应的函数对象。

9.8 Exceptions Are Classes Too 异常也是类

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

用户自定义异常也可以是类。利用这个机制可以创建可扩展的异常体系。

There are two new valid (semantic) forms for the `raise` statement:

以下是两种新的，有效的（语义上的）异常抛出形式，使用 `raise` 语句

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

第一种形式中， `instance` 必须是 `Class` 或其派生类的一个实例。 第二种形式是以下形式的简写

```
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around --- an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

发生的异常其类型如果是 `except` 子句中列出的类，或者是其派生 类，那么它们就是相符的（反过来说——发生的异常其类型如果是异常子句中列 出的类的基类，它们就不相符）。例如，以下代码会按顺序打印 B，C，D

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B --- the first matching except clause is triggered.

要注意的是如果异常子句的顺序颠倒过来（ `execpt B` 在最前），它就会打印 B，B，B——第一个匹配的异常被触发。

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function str().

打印一个异常类的错误信息时，先打印类名，然后是一个空格、一个冒号，然后 是用内置函数 str() 将类转换得到的完整字符串。

9.9 Iterators 迭代器

By now you have probably noticed that most container objects can be looped over using a for statement:

现在你可能注意到大多数容器对象都可以用 for 遍历

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the for statement calls iter() on the container object. The function returns an iterator object that defines the method next() which accesses elements in the container one at a time. When there are no more elements, next() raises a StopIteration exception which tells the for loop to terminate. This example shows how it all works:

这种形式的访问清晰、简洁、方便。迭代器的用法在 Python 中普遍而且统一。 在后台， for 语句在容器对象中调用 iter() 。 该函数返 回一个定义了 next() 方法的迭代器对象，它在容器中逐一访问元素。没 有后续的元素时， next() 抛出一个 StopIteration 异常通知 for 语句循环结束。以下是其工作原理的示例

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define a __iter__() method which returns an object with a next() method. If the class defines next(), then __iter__() can just return self:

了解了迭代器协议的后台机制，就可以很容易的给自己的类添加迭代器行为。定 义一个 `__iter__()` 方法，使其返回一个带有 `next()` 方法的对象。如果这个类 已经定义了 `next()` ，那么 `__iter__()` 只需要返回 self

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s
```

9.10 Generators 生成器

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

生成器 是创建迭代器的简单而强大的工具。它们写起来就像是正规的函 数，需要返回数据的时候使用 `yield` 语句。每次 `next()` 被 调用时，生成器回复它脱离的位置（它记忆语句最后一次执行的位置和所有的数 据值）。以下示例演示了生成器可以很简单的创建出来

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Anything that can be done with generators can also be done with class based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

前一节中描述了基于类的迭代器，它能作的每一件事生成器也能作到。因为自动 创建了 `__iter__()` 和 `next()` 方法，生成器显得如此简洁。

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

另一个关键的功能在于两次执行之间，局部变量和执行状态都自动的保存下来。 这使函数很容易写，而且比使用 `self.index` 和 `self.data` 之类的方 式更清晰。

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

除了创建和保存程序状态的自动方法，当发生器终结时，还会自动抛出 `StopIteration` 异常。综上所述，这些功能使得编写一个正规函数成为 创建迭代器的最简单方法。

9.11 Generator Expressions 生成器表达式

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

有时简单的生成器可以用简洁的方式调用，就像不带中括号的链表推导式。这些 表达式是为函数调用生成器而设计的。生成器表达式比完整的生成器定义更简 洁，但是没有那么多变，而且通常比等价的链表推导式更容易记。

Examples:

例如

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

CHAPTER

TEN

BRIEF TOUR OF THE STANDARD LIBRARY 标准库概览

10.1 Operating System Interface 操作系统接口

The os module provides dozens of functions for interacting with the operating system:

os 模块提供了不少与操作系统相关联的函数

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')
```

Be sure to use the import os style instead of from os import *. This will keep os.open() from shadowing the built-in open() function which operates much differently.

应该用 import os 风格而非 from os import * 。这样可以保证随操作系统 不同而有所变化的 os.open() 不会覆盖内置函数 open() 。 The built-in dir() and help() functions are useful as interactive aids for working with large modules like os:

在使用一些像 os 这样的大型模块时内置的 dir() 和 help() 函数非常有用

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the shutil module provides a higher level interface that is easier to use:

针对日常的文件和目录管理任务，:mod:shutil 模块提供了一个易于使用的高级接口

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2 File Wildcards 文件通配符

The glob module provides a function for making file lists from directory wildcard searches:

glob 模块提供了一个函数用于从目录通配符搜索中生成文件列表

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Command Line Arguments 命令行参数

Common utility scripts often need to process command line arguments. These arguments are stored in the sys module's argv attribute as a list. For instance the following output results from running python demo.py one two three at the command line:

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于 sys 模 块的 argv 变量。例如在命令行中执行 python demo.py one two three 后可 以得到以下输出结果

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

The getopt module processes sys.argv using the conventions of the Unix getopt() function. More powerful and flexible command line processing is provided by the argparse module.

getopt 模块使用 Unix getopt() 函处理 sys.argv 。更多的 复杂命令行处理由 optparse 模块提供。

10.4 Error Output Redirection and Program Termination 错误输出重定向和程序终止

The sys module also has attributes for stdin, stdout, and stderr. The latter is useful for emitting warnings and error messages to make them visible even when stdout has been redirected:

sys 还有 stdin ， stdout 和 stderr 属性，即使在 stdout 被重定向时，后者 也可以用于显示警告和错误信息

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use sys.exit().

大多脚本的定向终止都使用 sys.exit() 。

10.5 String Pattern Matching 字符串正则匹配

The re module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

re 模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正 则表达式提供了简洁、优化的解决方案

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

只需简单的操作时，字符串方法最好用，因为它们易读，又容易调试

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Mathematics 数学

The `math` module gives access to the underlying C library functions for floating point math:

`math` 模块为浮点运算提供了对底层C函数库的访问

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The `random` module provides tools for making random selections:

`random` 提供了生成随机数的工具

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

10.7 Internet Access 互联网访问

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from urls and `smtplib` for sending mail:

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处 理从 `urls` 接收的数据的 `urllib2` 以及用于发送电子邮件的 `smtplib`

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

(注意第二个例子需要在 localhost 运行一个邮件服务器。)

10.8 Dates and Times 日期和时间

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware. :

datetime 模块为日期和时间处理同时提供了简单和复杂的方法。支持日期和时 间算法的同时,实现的重点放在更有效的处理和格式化输出。该模块还支持时区 处理

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Data Compression 数据压缩

Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, zipfile and tarfile. :

以下模块直接支持通用的数据打包和压缩格式: zlib , gzip , bz2 , zipfile , 以及 tarfile

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Performance Measurement 性能度量

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提 供了一个度量工具，为这些问题提供了直接答案。

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The timeit module quickly demonstrates a modest performance advantage:

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多。 timeit 证明了后者更快一些

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to timeit`s fine level of granularity, the profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.

相对于 timeit 的细粒度，:mod:profile 和 pstats 模块提 供了针对更大代码块的时间度量工具。

10.11 Quality Control 质量控制

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试。

The doctest module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

doctest 模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测 试。测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。通过用户 提供的例子，它发展了文档，允许 doctest 模块确认代码的结果是否与文档一 致

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

The unittest module is not as effortless as the doctest module, but it allows a more comprehensive set of tests to be maintained in a separate file:

unittest 模块不像 doctest 模块那么容易使用，不过它可以在一个独立的文件 里提供一个更全面的测试集

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

10.12 Batteries Included 电池已备

Python has a ``batteries included'' philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

Python 体现了 "batteries included" 哲学 [1] 。Python 可以通过更大的包的来得 到应付各种复杂情况的强大能力，从这一点我们可以看出该思想的应用。例如：

- The xmlrpclib and SimpleXMLRPCServer modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.

  xmlrpclib 和 SimpleXMLRPCServer 模块实现了在琐碎的任务中调用远程过程。 尽管有这样的名字，其实用户不需要直接处理 XML ，也不需要这方面的知识。

- The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike smtplib and poplib which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.

  email 包是一个邮件消息管理库，可以处理 MIME 或其它基于 RFC 2822 的消 息文档。不同于实际发送和接收消息的 smtplib 和 poplib 模块，email 包 有一个用于构建或解析复杂消息结构（包括附件）以及实现互联网编码和头协 议的完整工具集。

- The xml.dom and xml.sax packages provide robust support for parsing this popular data interchange format. Likewise, the csv module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.

  xml.dom 和 xml.sax 包为流行的信息交换格式提供了强大的支持。同样， csv 模块支持在通用数据库格式中直接读写。综合起来，这些模块和包大大简 化了 Python 应用程序和其它工具之间的数据交换。

- Internationalization is supported by a number of modules including gettext, locale, and the codecs package.

  国际化由 gettext ， locale 和 codecs 包支持。

[1] batteries included 是一个商业用语，印在玩具之类的商品上，表示产品中附有电池，开箱可用。——译注

CHAPTER

ELEVEN

BRIEF TOUR OF THE STANDARD LIBRARY -- PART II 标准库概览 II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

第二部分包含了支持专业编程工作所需的更高级的模块，这些模块很少出现在小脚本中。

11.1 Output Formatting 输出格式

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

`repr` 提供了一个 `repr()` 的定制版本，以显示大型或深度嵌套的容器

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the ``pretty printer'' adds line breaks and indentation to more clearly reveal data structure:

`pprint` 模块给老手提供了一种解释器可读的方式深入控制内置和用户自定 义对象的打印。当输出超过一行的时候，"美化打印（pretty printer）" 添加 断行和标识符，使得数据结构显示的更清晰

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

`textwrap` 模块格式化文本段落以适应设定的屏宽

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
```

```
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The locale module accesses a database of culture specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

locale 模块按访问预定好的国家信息数据库。locale 的格式化函数属性集提供 了一个直接方式以分组标示格式化数字

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Templating 模板

The string module includes a versatile Template class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

string 提供了一个灵活多变的模版类 template ，使用它最 终用户可以用简单的进行编辑。这使用户可以在不进行改变的情况下定制他们的 应用程序。

The format uses placeholder names formed by $ with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing $$ creates a single escaped $:

格式使用 $ 为开头的 Python 合法标识（数字、字母和下划线）作为占位 符。占位符外面的大括号使它可以和其它的字符不加空格混在一起。 $$ 创 建一个单独的 $

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The substitute() method raises a KeyError when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the safe_substitute() method may be more appropriate --- it will leave placeholders unchanged if data is missing:

字典或者关键字参数中缺少某个占位符的时候 substitute() 方法抛出 KeyError 异常。在邮件-合并风格的应用程序中，用户提供的数据可能并不完整，也许用 safe-substitute() 方法更合适——如果数据不完整，它保留未改动的占位符

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

模板子类可以指定一个定制分隔符。例如，图像浏览器的批量命名工具可能选用 百分号作为表示当前日期、图像序列号或文件格式的占位符

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '{0} --> {1}'.format(filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

另一个应用是将多样化的输出格式细节从程序逻辑中分离出来。这使得为 XML 文件，纯文本报表，HTML web 报表定制替换模版成为可能。

11.3 Working with Binary Data Record Layouts 使用二进制记录层

The struct module provides pack() and unpack() functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the zipfile module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

struct 模块提供 pack() 和 unpack() 函数用于变长二进制记录格式。以下示 例显示了如何通过 ZIP 文件的头信息（压缩代码中的 "H" 和 "L" 分别传递二和 四字节无符号整数）， "<" 标识了它们以标准尺寸和低地址低字节存储

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
```

```
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size     # skip to the next header
```

11.4 Multi-threading 多线程

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

线程是一个分离无顺序依赖关系任务的技术。在某些任务运行于后台的时候应用 程序会变得迟缓，线程可以提升其速度。一个有关的用途是在 I/O 的同时其它线 程可以并行计算。

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

下面的代码显示了高级模块 `threading` 如何在主程序运行的同时运行任 务

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

多线程应用程序最重要的挑战是在协调线程共享的数据和其它资源。另外，线程 模块提供了几个基本的同步方式如锁、事件，条件变量和信号量。

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the Queue module to feed that thread with requests from other threads. Applications using Queue.Queue objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

尽管工具很强大，微小的设计错误也可能造成难以挽回的故障。因此，更好的方 法是将所有的资源访问集中到一个独立的线程中，然后使用 Queue 模块调度该 线程相应其它线程的请求。应用程序使用 Queue.Queue 对象可以让内部线程通信和协 调更容易设计，更可读，更可靠。

11.5 Logging 日志

The logging module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to sys.stderr:

logging 模块提供了完整和灵活的日志系统。它最简单的用法是记录信息并发送 到一个文件或 sys.stderr

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

输出如下

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: DEBUG, INFO, WARNING, ERROR, and CRITICAL.

默认情况下捕获信息和调试消息并将输出发送到标准错误流。其它可选的路由信 息方式通过email，数据报文，socket或者HTTP Server。基于消息属性，新的过 滤器可以选择不同的路由： DEBUG, INFO ， WARNING ， ERROR 和 CRITICAL 。

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

日志系统可以直接在 Python 代码中定制，也可以不经过应用程序直接在一个用户可编辑的配置文件中加载。

11.6 Weak References 弱引用

Python does automatic memory management (reference counting for most objects and garbage collection to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

Python 自动进行内存管理（对大多数的对象进行引用计数和垃圾回收—— garbage collection ——以循环利用）在最后一个引用消失后，内存会 很快释放。

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The weakref module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

这个工作方式对大多数应用程序工作良好，但是偶尔会需要跟踪对象来做一些事。 不幸的是，仅仅为跟踪它们创建引用也会使其长期存在。 weakref 模块 提供了不用创建引用的跟踪对象工具，一旦对象不再存在，它自动从弱引用表上 删除并触发回调。典型的应用包括捕获难以构造的对象

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Tools for Working with Lists 列表工具

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

很多数据结构可能会用到内置列表类型。然而，有时可能需要不同性能代价的实现。

The array module provides an array() object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

array 模块提供了一个类似列表的 array 对象，它仅仅是存储数据，更为紧凑。 以下的示例演示了一个存储双字节无符号整数的数组（类型编码 "H" ）而非存储 16 字节 Python 整数对象的普通正规列表

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The collections module provides a deque() object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

collections 模块提供了类似列表的 deque() 对象，它从左边添加（append） 和弹出（pop）更快，但是在内部查询更慢。这些对象更适用于队列实现和广度 优先的树搜索

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the bisect module with functions for manipulating sorted lists:

除了链表的替代实现，该库还提供了 bisect 这样的模块以操作存储链 表

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The heapq module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

heapq 提供了基于正规链表的堆实现。最小的值总是保持在0点。这在希 望循环访问最小元素但是不想执行完整堆排序的时候非常有用

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Decimal Floating Point Arithmetic 十进制浮点数算法

The decimal module offers a Decimal datatype for decimal floating point arithmetic. Compared to the built-in float implementation of binary floating point, the class is especially helpful for

decimal 模块提供了一个 Decimal 数据类型用于浮点数计算。相比内置 的二进制浮点数实现 float ，这个类型有助于

- financial applications and other uses which require exact decimal representation,

  金融应用和其它需要精确十进制表达的场合，

- control over precision,

  控制精度，

- control over rounding to meet legal or regulatory requirements,

  控制舍入以适应法律或者规定要求，

- tracking of significant decimal places, or

  确保十进制数位精度，或者

- applications where the user expects the results to match calculations done by hand.

  用户希望计算结果与手算相符的场合。

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

例如，计算 70 分电话费的 5% 税计算，十进制浮点数和二进制浮点数计算结果 的差别如下。如果在分值上舍入，这个差别就很重要了

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

The Decimal result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Decimal 的结果总是保有结尾的 0，自动从两位精度延伸到4位。 Decimal 重现了手工的数学运算，这就确保了二进制浮点数无法精确保有的数据 精度。

Exact representation enables the Decimal class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

高精度使 Decimal 可以执行二进制浮点数无法进行的模运算和等值测试。

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

The decimal module provides arithmetic with as much precision as needed:

decimal 提供了必须的高精度算法

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CHAPTER

TWELVE

WHAT NOW? 接下来?

Reading this tutorial has probably reinforced your interest in using Python --- you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

读过这本指南应该会让你有兴趣使用 Python —— 可能你已经期待着用 Python 解决你的实际问题了。可以在哪里进行一步学习？

This tutorial is part of Python's documentation set. Some other documents in the set are:

入门指南是 Python 文档集的一部分。其中的另一些文档包括：

- library-index:

  You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a lot of additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what's available.

  应该浏览一下这份文档，它为标准库中的类型、函数和模块提供了完整（尽管很 简略）的参考资料。标准的 Python 发布版包括了 大量 的附加模块。其中 有针对读取 Unix 邮箱、接收 HTTP 文档、生成随机数、解析命令行选项、写 CGI 程序、压缩数据以及很多其它任务的模块。略读一下库参考会给你很多解决问 题的思路。

- install-index explains how to install external modules written by other Python users.

  install-index 展示了如何安装其他 Python 用户编写的附加模块。

- reference-index: A detailed explanation of Python's syntax and semantics. It's heavy reading, but is useful as a complete guide to the language itself.

  reference-index 詳細说明了 Python 语法和语义。它读起来很累，不 过对于语言本身，有份完整的手册很有用。

More Python resources 其它 Python 资源:

- http://www.python.org: The major Python Web site. It contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location.

  http://wwww.python.org: Python 官方网站。它包含代码、文档和 Web 上与 Python 有关的页面链接。该网站镜像于全世界的几处其它问题，类似欧洲、 日本和澳大利亚。镜像可能会比主站快，这取决于你的地理位置。

- http://docs.python.org: Fast access to Python's documentation.

http://docs.python.org: 快速访问 Python 的文档。

- http://pypi.python.org: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.

  http://pypi.python.org: Python 包索引，以前昵称为奶酪店，索引了可供 下载的，用户创建的 Python 模块。如果你发布了代码，可以注册到这里，这 样别人可以找到它。

- http://aspn.activestate.com/ASPN/Python/Cookbook/: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

  http://aspn.activestate.com/ASPN/python/Cookbook/: Python 食谱是 大量的示例代码、大型的集合，和有用的脚本。值得关注的是这次资源已经结 集成书，名为《Python 食谱》（O'Reilly & Associates, ISBN 0-596-00797-3。）

For Python-related questions and problem reports, you can post to the newsgroup *comp.lang.python*, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions <http://www.python.org/doc/faq/> [1] (also called the FAQ), or look for it in the Misc/ directory of the Python source distribution. Mailing list archives are available at http://mail.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

与 Python 有关的问题，以及问题报告，可以发到新闻组 *comp.lang.python* ，或者发送到邮件组 python-list@python.org 。新闻组和邮件组是开放的，所以发送的消息可以自 动的跟到另一个之后。每天有超过 120 个投递（高峰时有数百）,提问（以及回 答）问题，为新功能提建议，发布新模块。在发信之前，请查阅“常见问题集 ” <http://www.python.org/doc/faq/>` [2] （亦称 FAQ），或者在 Python 源码发 布包的 Misc/ 目录中查阅。邮件组也可以在 http://mail.python.org/pipermail/ 访问。FAQ回答了很多被反复提到的问题，很 可能已经解答了你的问题。

[1] Postings figure based on average of last six months activity as reported by www.egroups.com; Jan. 2000 - June 2000: 21272 msgs / 182 days = 116.9 msgs / day and steadily increasing. (XXX up to date figures?)

[2] 最近六个月的提交数在 www.egroups.com 有报告；2000年六月：21272 封 / 182 天 = 116.9 封／天，并且还在平稳的上升。（现在是多少？）

CHAPTER

THIRTEEN

INTERACTIVE INPUT EDITING AND HISTORY SUBSTITUTION

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

有些版本的 Python 解释器支持输入行编辑和历史回溯，类似 Korn shell 和 GNU bash shell 的功能。这是通过 GNU Readline 库实现的。它支持 Emacs 风格和 vi 风格的编辑。这个库有它自己的文档，在此不重复了。不过，基本的 东西很容易演示。交互式编辑和历史查阅在 Unix 和 Cygwin 版中是可选项。

This chapter does not document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

本章 不是 马克 哈密尔顿的 PythonWin 包和随 Python 发布的基于 TK 的 IDLE 环境的文档。 NT 系统和其它 DOS、Windows 系统上的 DOS 窗中的命令行 历史回调，属于另一个话题。

13.1 Line Editing 行编辑

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: C-A (Control-A) moves the cursor to the beginning of the line, C-E to the end, C-B moves it one position to the left, C-F to the right. Backspace erases the character to the left of the cursor, C-D the character to its right. C-K kills (erases) the rest of the line to the right of the cursor, C-Y yanks back the last killed string. C-underscore undoes the last change you made; it can be repeated for cumulative effect.

如果支持，无论解释器打印主提示符或从属提示符，行编辑都会激活。当前行可 以用 Emacs 风格的快捷键编辑。其中最重要的是： C-A （Control-A） 将光标移动到行首， :kbd:C-E 移动到行尾， C-B 向左移一个字 符， C-F 向右移一位。退格向左删除一个符串， C-D 向右删除 一个字符。 C-K 删掉光标右边直到行尾的所有字符， C-Y 将最 后一次删除的字符串粘贴到光标位置。 C-underscore （underscores 即下划线，译注）撤销最后一次修改，它可以因积累作用重复。

13.2 History Substitution 历史回溯

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. C-P moves one line up (back) in the history buffer, C-N moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the Return key passes the current line to the interpreter. C-R starts an incremental reverse search; C-S starts a forward search.

历史代替可以工作。所有非空的输入行都被保存在历史缓存中，获得一个新的提 示符的时候，你处于这个缓存的最底的空行。 C-P 在历史缓存中上溯一 行， C-N 向下移一行。历史缓存中的任一行都可以编辑；按下 :kbd:Return 键时将当前行传入解释器。 C-R 开始一个增量向前搜 索；:kbd:C-S 开始一个向后搜索。

13.3 Key Bindings 快捷键绑定

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called ~/.inputrc. Key bindings have the form :

Readline 库的快捷键绑定和其它一些参数可以通过名为 ~/.inputrc 的初始化文件的替换命名来定制。快捷键绑定如下形式

```
key-name: function-name
```

or :

或

```
"string": function-name
```

and options can be set with :

选项可以如下设置

```
set option-name value
```

For example:

例如

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for Tab in Python is to insert a Tab character instead of Readline's default filename completion function. If you insist, you can override this by putting :

需要注意的是 Python 中默认 Tab 绑定为插入一个 Tab 字符 而不是 Readline 库的默认文件名完成函数，如果你想用这个，可以将以下内容 插入

```
Tab: complete
```

in your ~/.inputrc. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using Tab for that purpose.)

到你的 ~/.inputrc 中来覆盖它。（当然，如果你真的把 Tab 设置成这样，就很难在后继行中插入缩进。） Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file: [1] :

自动完成变量和模块名也可以激活生效。要使之在解释器交互模式中可用，在你 的启动文件中加入下面内容：[2]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the Tab key to the completion function, so hitting the Tab key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as string.a, it will evaluate the expression up to the final '.' and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a __getattr__() method is part of the expression.

这个操作将 Tab 绑定到完成函数，故按 Tab 键两次会给出建议 的完成内容；它查找 Python 命名、当前的局部变量、有效的模块名。对于类 似 string.a 这样的文件名，它会解析 '.' 相关的表达式，从返回的 结果对象中获取属性，以提供完成建议。需要注意的是，如果对象的 __getattr__() 方法是此表达式的一部分，这可能会执行应用程序定 义代码。

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as os, which turn out to be needed in most sessions with the interpreter. :

更有用的初始化文件可能是下面这个例子这样的。要注意一旦创建的名字没用 了，它会删掉它们；因为初始化文件作为解释命令与之在同一个命名空间执行， 在交互环境中删除命名带来了边际效应。可能你发现了它体贴的保留了一些导入 模块，类似 os ，在解释器的大多数使用场合中都会用到它们。

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=/home/user/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
```

[1] Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter.

[2] 启动交互解释器时，Python 可以执行 PYTHONSTARTUP 环境变量所指 定的文件内容。

```
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4 Alternatives to the Interactive Interpreter 其它交互式解释器

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

跟早先版本的解释器比，现在已经有了很大的进步。不过，还是有些期待没有完 成：它应该在后继行中优美的提供缩进（解释器知道下一行是否需要缩 进）建议。完成机制可以使用解释器的符号表。命名检查（或进一步建议）匹配 括号、引号等等。

One alternative enhanced interactive interpreter that has been around for quite some time is IPython, which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython.

另有一个强化交互式解释器已经存在一段时间了，它就是 `IPython`， 它支持 tab 完成，对象浏览和高级历史管理。它也可以完全定制或嵌入到其它 应用程序中。另一个类似的强化交互环境是 `bpython`。

CHAPTER

FOURTEEN

FLOATING POINT ARITHMETIC: ISSUES AND LIMITATIONS 浮点数算法：争议和限制

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction :

浮点数在计算机中表达为二进制（binary）小数。例如：十进制小数

```
0.125
```

has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction :

是 1/10 + 2/100 + 5/1000 的值，同样二进制小数

```
0.001
```

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

是 0/2 + 0/4 + 1/8。这两个数值相同。唯一的实质区别是第一个写为十进制小 数记法，第二个是二进制。

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

遗憾的是，大多数十进制小数不能精确的表达二进制小数。

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

这个问题更早的时候首先在十进制中发现。考虑小数形式的 1/3 ，你可以来个 十进制的近似值

```
0.3
```

or, better, :

或者更进一步的，

```
0.33
```

or, better, :

或者，更进一步的

```
0.333
```

and so on. No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.

诸如此类。如果你写多少位，这个结果永远不是精确的 1/3 ，但是可以无限接 近 1/3 。

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction :

同样，无论在二进制中写多少位，十进制数 0.1 都不能精确表达为二进制小数。 二进制来表达 1/10 是一个无限循环小数

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation. This is why you see things like:

在任意无限位数值中中止，你可以得到一个近似，这就是为什么你会看见这个

```
>>> 0.1
0.10000000000000001
```

On most machines today, that is what you'll see if you enter 0.1 at a Python prompt. You may not, though, because the number of bits used by the hardware to store floating-point values can vary across machines, and Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display :

在今天的大多数机器上，如果你在 Python 提示符后输入 0.1，就会看到上面的 内容。当然，也许你看到的不一样，因为不同的机器存储浮点数数值位的硬件会有区 别，Python 只打印十进制小数以二进制存储在机器中的近似值的十进制近似表示。 在大多数机器上，如果 Python 打印 0.1 的二进制存储的真正十进制值，应该 显示为这样！

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

instead! The Python prompt uses the built-in `repr()` function to obtain a string version of everything it displays. For floats, `repr(float)` rounds the true decimal value to 17 significant digits, giving :

Python 使用内置的 `repr()` 函数获取它要显示的每一个对象的字符串版 本。对于浮点数， `repr(float)` 将真正的十进制值处理为十七位精度，得 到

```
0.10000000000000001
```

`repr(float)` produces 17 significant digits because it turns out that's enough (on most machines) so that `eval(repr(x)) == x` exactly for all finite floats x, but rounding to 16 digits is not enough to make that true.

`repr(float)` 生成 17 位精度，这是因为它已经足够了（在大多数机器上）。 依此 `eval(repr(x)) == x` 可以精确的应用到所有的无限浮点数 x ，但 是 16 位的话就不够，不一定得到 true。

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not display the difference by default, or in all output modes).

需要注意的是这在二进制浮点数是非常自然的：它不是 Python 的bug，也不是 你的代码的 bug。你会看到只要你的硬件支持浮点数算法，所有的语言都会有这 个现象（尽管有些语言可能默认或完全不 显示 这个差异）。

Python's built-in str() function produces only 12 significant digits, and you may wish to use that instead. It's unusual for eval(str(x)) to reproduce x, but the output may be more pleasant to look at:

Python 的内置函数 str() 只生成 12位精度。你可能更希望用它。通常 它用形如 eval(str(x)) 来重现 x ，而看起来更合意

```
>>> print str(0.1)
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the display of the true machine value.

认识到这个幻觉的真相很重要：机器不能精确表达 1/10，你可以简单的截断 显示 真正的机器值。

Other surprises follow from this one. For example, after seeing :

这里还有另一个惊奇之处。例如，下面

```
>>> 0.1
0.10000000000000001
```

you may be tempted to use the round() function to chop it back to the single digit you expect. But that makes no difference:

你可能受鼓动去尝试 :func:round 函数来截断这个数，使其回到你期待的精 度，但是结果有些出乎意料

```
>>> round(0.1, 1)
0.10000000000000001
```

The problem is that the binary floating-point value stored for ``0.1'' was already the best possible binary approximation to 1/10, so trying to round it again can't make it better: it was already as good as it gets.

这个问题在于存储“０.１”的浮点值已经达到1/10的最佳精度了，所以尝试截 断它不能改善：它已经尽可能的好了。

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

另一个影响是因为 0.1 不能精确的表达 1/10，对10个 0.1 的值求和不能精确 的得到 1.0，即

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

Binary floating-point arithmetic holds many surprises like this. The problem with ``0.1'' is explained in precise detail below, in the ``Representation Error'' section. See The Perils of Floating Point for a more complete account of other common surprises.

浮点数据算法产生了很多诸如此类的惊奇。在“表现错误”一节中，这个 “0.1”问题详细表达了精度问问题。更完整的其它常见的惊奇请参见“浮点数 的危害 <http://www.lahey.com/float.html> ”。

As that says near the end, ``there are no easy answers.'' Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

最后我要说，“没有简单的答案”。还是不要过度的敌视浮点数！Python 浮点 数操作的错误来自于浮点数硬件，大多数机器上同类的问题每次计算误差不超过 2**53 分之一。对于大多数任务这已经足够让人满意了。但是你要在心中记住这不是十 进制算法，每个浮点数计算可能会带来一个新的精度错误。

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method's format specifiers in formatstrings.

问题已经存在了，对于大多数偶发的浮点数错误，你应该比对你期待的最终显示结果 是否符合你的期待。`str()` 通常够用了，完全的控制参见 formatstrings 中 `str.format()` 方法的格式化方式。

14.1 Representation Error 表达错误

This section explains the ``0.1'' example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

这一节详细说明“0.1”示例，教你怎样自己去精确的分析此类案例。假设这里 你已经对浮点数表示有基本的了解。

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

Representation error 提及事实上有些（实际是大多数）十进制小数不 能精确的表示为二进制小数。这是 Python （或 Perl，C，C++，Java，Fortran 以及其它很多）语言往往不能按你期待的样子显示十进制数值的根本原因

```
>>> 0.1
0.10000000000000001
```

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 ``double precision''. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form J/2**N where J is an integer containing exactly 53 bits. Rewriting :

这是为什么？ 1/10 不能精确的表示为二进制小数。大多数今天的机器（2000年 十一月）使用 IEEE-754 浮点数算法，大多数平台上 Python 将浮点数映射为 IEEE-754 “双精度浮点数”。754 双精度包含 53 位精度，所以计算机努力将 输入的 0.1 转为 J/2**N 最接近的二进制小数。*J* 是一个 53 位的整 数。改写

```
1 / 10 ~= J / (2**N)
```

as :

为 ：

```
J ~= 2**N / 10
```

and recalling that J has exactly 53 bits (is >= 2**52 but < 2**53), the best value for N is 56:

J 重现时正是 53 位（是 >= 2**52 而非 < 2**53 ），N 的最佳值是 56

```
>>> 2**52
4503599627370496L
>>> 2**53
9007199254740992L
```

```
>>> 2**56/10
7205759403792793L
```

That is, 56 is the only value for N that leaves J with exactly 53 bits. The best possible value for J is then that quotient rounded:

因此，56 是保持 J 精度的唯一 N 值。 J 最好的近似值是整除的商

```
>>> q, r = divmod(2**56, 10)
>>> r
6L
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

因为余数大于 10 的一半，最好的近似是取上界

```
>>> q+1
7205759403792794L
```

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or :

因此在 754 双精度中 1/10 最好的近似值是是 2**56，或

```
7205759403792794 / 72057594037927936
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be exactly 1/10!

要注意因为我们向上舍入，它其实比 1/10 稍大一点点。如果我们没有向上舍 入，它会比 1/10 稍小一点。但是没办法让它 恰好 是 1/10！

So the computer never ``sees'' 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

所以计算机永远也不 “知道” 1/10：它遇到上面这个小数，给出它所能得到的 最佳的 754 双精度实数

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

如果我们用 10**30 除这个小数，会看到它最大30位（截断后的）的十进制值

```
>>> 7205759403792794 * 10**30 / 2**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. Rounding that to 17 significant digits gives the 0.10000000000000001 that Python displays (well, will display on any 754-conforming platform that does best-possible input and output conversions in its C library --- yours may not!).

这表示存储在计算机中的实际值近似等于十进制值 0.100000000000000005551115123125。 Python 显示时取 17 位精度为 0.10000000000000001（是的，在任何符合754的平台上，都会由其C库转换为这 个最佳近似——你的可能不一样！）。